# ACTIONSCRIPT™ 3.0 编程



ActionScript™ 3.0 编程



Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA

# 目录

# 关于本手册

本手册为在 ActionScript™ 3.0 中开发应用程序提供了基础。为了充分理解所介绍的理念和技巧，您应已熟悉了一般的编程概念，如数据类型、变量、循环和函数。您还应了解面向对象编程的基本概念，如类和继承。如果以前掌握了 ActionScript 1.0 或 ActionScript 2.0 知识，则会非常有帮助，但这并不是必需的。

目录



# 使用本手册

本手册中的章节划分为以下逻辑组，以帮助您方便地查找 ActionScript 文档的相关部分：

| 章节 | 描述 |
|---|---|
| 第 1-4 章， ActionScript 编程概述 | 讨论 ActionScript 3.0 核心概念，其中包括语言语法、语句和运算符、ECMAScript 第 4 版语言规范草案、面向对象的 ActionScript 编程以及管理 Adobe® Flash® Player 9 显示列表中的显示对象的新方法。 |
| 第 5-10 章， ActionScript 3.0 核心数据类型和类 | 介绍 ActionScript 3.0 中的顶级数据类型 （也是 ECMAScript 规范草案的一部分）。 |
| 第 11-26 章， Flash Player API | 介绍在特定于 Adobe Flash Player 9 的包和类中实现的重要功能，其中包括事件处理、网络和通信、文件输入和输出、外部接口、应用程序安全模型等。 |

本手册还包含许多范例文件，这些文件演示重要类或常用类的应用程序编程概念。范例文件是以打包形式提供的，这样便于轻松加载和与 Adobe® Flash® CS3 Professional 一起使用。这些范例文件还可能包含包装文件。不过，核心范例代码是纯粹的 ActionScript 3.0，您可以在您喜爱的任何开发环境中使用。

可以使用多种方法来编写并编译 ActionScript 3.0，其中包括：

- 使用 Adobe Flex Builder 2 开发环境
- 使用任何文本编辑器和命令行编译器，如随 Flex Builder 2 提供的编辑器和编译器
- 使用 Adobe® Flash® CS3 Professional 创作工具

有关 ActionScript 开发环境的详细信息，请参阅第 1 章 “ActionScript 3.0 简介”。

要理解本手册中的代码范例，您并不需要以前具有使用 ActionScript 集成开发环境的经验，如 Flex Builder 或 Flash 创作工具。但是，您需要参考这些工具的文档，以了解如何使用它们来编写并编译 ActionScript 3.0 代码。有关详细信息，请参阅第 14 页的 “访问 ActionScript 文档”。

# 访问 ActionScript 文档

因为本手册重点介绍 ActionScript 3.0（一种内容丰富且功能强大的面向对象编程语言），所以，它并未详细介绍特定工具或服务器体系结构内的应用程序开发过程或工作流程。因此，在设计、开发、测试和部署 ActionScript 3.0 应用程序时，除了《ActionScript 3.0 编程》外，您还需要查阅其它文档来源。

## ActionScript 3.0 文档

本手册向您介绍 ActionScript 3.0 编程语言的概念，并提供实现详细信息以及阐释重要语言功能的范例。但是，本手册不是完整的语言参考。有关完整的语言参考，请参阅《ActionScript 3.0 语言和组件参考》，该手册描述了语言中的每一个类、方法、属性和事件。《ActionScript 3.0 语言和组件参考》提供了关于核心语言、 Flash 组件 （在 fl 包中）和 Flash Player API （在 Flash 包中）的详细参考信息。

## Flash 文档

使用 Flash 开发环境时，可能需要参考以下这些手册：

| 书籍 | 描述 |
|---|---|
| 《使用 Flash》 | 描述如何在 Flash 创作环境中开发动态 Web 应用程序 |
| 《ActionScript 3.0 编程》 | 描述 ActionScript 3.0 语言和核心 Flash Player API 的具体用法 |
| 《ActionScript 3.0 语言和组件参考》 | 提供 Flash 组件和 ActionScript 3.0 API 的语法、用法和代码示例 |
| 《使用 ActionScript 3.0 组件》 | 详细说明如何使用组件开发 Flash 应用程序 |

| 书籍 | 描述 |
| --- | --- |
| 《学习 Adobe Flash 中的 ActionScript 2.0》 | 提供 ActionScript 2.0 的语法概述并介绍在处理不同对象类型时如何使用 ActionScript 2.0 |
| 《ActionScript 2.0 语言参考》 | 提供 Flash 组件和 ActionScript 2.0 API 的语法、用法和代码示例 |
| 《使用 ActionScript 2.0 组件》 | 详细介绍如何使用 ActionScript 2.0 组件来开发 Flash 应用程序 |
| 《ActionScript 2.0 组件语言参考》 | 描述第 2 版 Adobe 组件体系结构中提供的每个组件以及它的 API |
| 《扩展 Flash》 | 描述 JavaScript API 中提供的对象、方法和属性 |
| 《Flash Lite 2.x 快速入门》 | 介绍如何使用 Adobe® Flash® Lite™ 2.x 来开发应用程序，并提供可以与 Flash Lite 2.x 一起使用的 ActionScript 功能的语法、用法和代码示例 |
| 《开发 Flash Lite 2.x 应用程序》 | 介绍如何开发 Flash Lite 2.x 应用程序 |
| 《Flash Lite 2.x ActionScript 简介》 | 介绍如何使用 Flash Lite 2.x 来开发应用程序，并描述 Flash Lite 2.x 开发人员可以使用的所有 ActionScript 功能 |
| 《Flash Lite 2.x ActionScript 语言参考》 | 提供可以在 Flash Lite 2.x 中使用的 ActionScript 2.0 API 的语法、用法和代码示例 |
| 《Flash Lite 1.x 快速入门》 | 介绍 Flash Lite 1.x 并描述如何使用 Adobe® Device Central CS3 模拟器测试您的内容 |
| 《开发 Flash Lite 1.x 应用程序》 | 描述如何使用 Flash Lite 1.x 来开发用于移动设备的应用程序 |
| 《学习 Flash Lite 1.x ActionScript》 | 介绍如何在 Flash Lite 1.x 应用程序中使用 ActionScript，并描述可用于 Flash Lite 1.x 的所有 ActionScript 功能 |
| 《Flash Lite 1.x ActionScript 语言参考》 | 提供可用于 Flash Lite 1.x 的 ActionScript 元素的语法和用法 |

# ActionScript 学习资源

除了这些手册中的内容以外， Adobe 还在 Adobe 开发人员中心和 Adobe 设计中心上提供定期更新的文章、设计思路和示例。

## Adobe 开发人员中心

Adobe 开发人员中心提供有关 ActionScript 的最新信息、有关实际应用程序开发的文章以及新出现的重要问题的相关信息。开发人员中心的网址为 www.adobe.com/devnet/。

## Adobe 设计中心

了解数字设计和动画图形方面的最新动态。您可以从中浏览主要艺术家的作品，了解新的设计趋势以及通过教程、重要工作流程和高级技巧来提高您的技能。请每月访问两次该网站以了解最新的教程和文章以及充满灵感的图片作品。设计中心的网址为 www.adobe.com/designcenter/。

第 1 章

# ActionScript 3.0 简介

1

本章概述了最新且最具创新性的 ActionScript 版本，即 ActionScript 3.0。

## 目录



## 关于 ActionScript

ActionScript 是针对 Adobe Flash Player 运行时环境的编程语言，它在 Flash 内容和应用程序中实现了交互性、数据处理以及其它许多功能。

ActionScript 是由 Flash Player 中的 ActionScript 虚拟机 (AVM) 来执行的。 ActionScript 代码通常被编译器编译成 “字节码格式”（一种由计算机编写且能够为计算机所理解的编程语言），如 Adobe Flash CS3 Professional 或 Adobe® Flex™ Builder™ 的内置编译器或 Adobe® Flex™ SDK 和 Flex™ Data Services 中提供的编译器。字节码嵌入 SWF 文件中，SWF 文件由运行时环境 Flash Player 执行。

ActionScript 3.0 提供了可靠的编程模型，具备面向对象编程的基本知识的开发人员对此模型会感到似曾相识。 ActionScript 3.0 中的一些主要功能包括：

- 一个新增的 ActionScript 虚拟机，称为 AVM2，它使用全新的字节码指令集，可使性能显著提高
- 一个更为先进的编译器代码库，它更为严格地遵循 ECMAScript (ECMA 262) 标准，并且相对于早期的编译器版本，可执行更深入的优化
- 一个扩展并改进的应用程序编程接口 (API)，拥有对对象的低级控制和真正意义上的面向对象的模型
- 一种基于即将发布的 ECMAScript (ECMA-262) 第 4 版草案语言规范的核心语言

- 一个基于 ECMAScript for XML (E4X) 规范 （ECMA-357 第 2 版）的 XML API。E4X 是 ECMAScript 的一种语言扩展，它将 XML 添加为语言的本机数据类型。
- 一个基于文档对象模型 (DOM) 第 3 级事件规范的事件模型

# ActionScript 3.0 的优点

ActionScript 3.0 的脚本编写功能超越了 ActionScript 的早期版本。它旨在方便创建拥有大型数据集和面向对象的可重用代码库的高度复杂应用程序。虽然 ActionScript 3.0 对于在 Adobe Flash Player 9 中运行的内容并不是必需的，但它使用新型的虚拟机 AVM2 实现了性能的改善。 ActionScript 3.0 代码的执行速度可以比旧式 ActionScript 代码快 10 倍。

旧版本的 ActionScript 虚拟机 AVM1 执行 ActionScript 1.0 和 ActionScript 2.0 代码。为了向后兼容现有内容和旧内容， Flash Player 9 支持 AVM1。有关详细信息，请参阅第 21 页的 “与早期版本的兼容性”。

# ActionScript 3.0 中的新增功能

虽然 ActionScript 3.0 包含 ActionScript 编程人员所熟悉的许多类和功能，但 ActionScript 3.0 在架构和概念上是区别于早期的 ActionScript 版本的。ActionScript 3.0 中的改进部分包括新增的核心语言功能，以及能够更好地控制低级对象的改进 Flash Player API。

## 核心语言功能

核心语言定义编程语言的基本构造块，例如语句、表达式、条件、循环和类型。ActionScript 3.0 包含许多加速开发过程的新功能。

### 运行时异常

ActionScript 3.0 报告的错误情形比早期的 ActionScript 版本多。运行时异常用于常见的错误情形，可改善调试体验并使您能够开发可以可靠地处理错误的应用程序。运行时错误可提供带有源文件和行号信息注释的堆栈跟踪，以帮助您快速定位错误。

### 运行时类型

在 ActionScript 2.0 中，类型注释主要是为开发人员提供帮助；在运行时，所有值的类型都是动态指定的。在 ActionScript 3.0 中，类型信息在运行时保留，并可用于多种目的。Flash Player 9 执行运行时类型检查，增强了系统的类型安全性。类型信息还可用于以本机形式表示变量，从而提高了性能并减少了内存使用量。

## 密封类

ActionScript 3.0 引入了密封类的概念。密封类只能拥有在编译时定义的固定的一组属性和方法；不能添加其它属性和方法。这使得编译时的检查更为严格，从而导致程序更可靠。由于不要求每个对象实例都有一个内部哈希表，因此还提高了内存的使用率。还可以通过使用 `dynamic` 关键字来实现动态类。默认情况下， ActionScript 3.0 中的所有类都是密封的，但可以使用 `dynamic` 关键字将其声明为动态类。

## 闭包方法

ActionScript 3.0 使闭包方法可以自动记起它的原始对象实例。此功能对于事件处理非常有用。在 ActionScript 2.0 中，闭包方法无法记起它是从哪个对象实例提取的，所以在调用闭包方法时将导致意外的行为。mx.utils.Delegate 类是一种常用的解决方法，但已不再需要。

# ECMAScript for XML (E4X)

ActionScript 3.0 实现了 ECMAScript for XML (E4X)，后者最近被标准化为 ECMA-357。E4X 提供一组用于操作 XML 的自然流畅的语言构造。与传统的 XML 分析 API 不同，使用 E4X 的 XML 就像该语言的本机数据类型一样执行。E4X 通过大大减少所需代码的数量来简化操作 XML 的应用程序的开发。有关 ActionScript 3.0 实现的 E4X 的详细信息，请参阅第 293 页的第 11 章 “处理 XML”。

要查看 ECMA 的 E4X 规范，请访问 www.ecma-international.org。

## 正则表达式

ActionScript 3.0 包括对正则表达式的固有支持，因此您可以快速搜索并操作字符串。由于在 ECMAScript (ECMA-262) 第 3 版语言规范中对正则表达式进行了定义，因此 ActionScript 3.0 实现了对正则表达式的支持。

## 命名空间

命名空间与用于控制声明（`public`、`private`、`protected`）的可见性的传统访问说明符类似。它们的工作方式与名称由您指定的自定义访问说明符类似。命名空间使用统一资源标识符 (URI) 以避免冲突，而且在您使用 E4X 时还用于表示 XML 命名空间。

## 新基元类型

ActionScript 2.0 拥有单一数值类型 Number，它是一种双精度浮点数。 ActionScript 3.0 包含 int 和 uint 类型。int 类型是一个带符号的 32 位整数，它使 ActionScript 代码可充分利用 CPU 的快速处理整数数学运算的能力。 int 类型对使用整数的循环计数器和变量都非常有用。uint 类型是无符号的 32 位整数类型，可用于 RGB 颜色值、字节计数和其它方面。

## Flash Player API 功能

ActionScript 3.0 中的 Flash Player API 包含许多允许您在低级别控制对象的新类。语言的体系结构是全新的并且更加直观。由于需要在这里详细介绍的新类实在太多，因此以下各节将着重介绍一些重要的更改。

### DOM3 事件模型

文档对象模型第 3 级事件模型 (DOM3) 提供了一种生成并处理事件消息的标准方法，以使应用程序中的对象可以进行交互和通信，同时保持自身的状态并响应更改。通过采用万维网联盟 DOM 第 3 级事件规范，该模型提供了一种比早期的 ActionScript 版本中所用的事件系统更清楚、更有效的机制。

事件和错误事件都位于 flash.events 包中。 Flash 组件框架使用的事件模型与 Flash Player API 相同，因此事件系统在整个 Flash 平台中是统一的。

### 显示列表 API

用于访问 Flash Player 显示列表的 API（包含 Flash 应用程序中的所有可视元素的树）由处理 Flash 中的可视基元的类组成。

新增的 Sprite 类是一个轻型构造块，它类似于 MovieClip 类，但更适合作为 UI 组件的基类。新增的 Shape 类表示原始的矢量形状。可以使用 `new` 运算符很自然地实例化这些类，并可以随时动态地重新指定其父类。

现在，深度管理是自动执行的并且已内置于 Flash Player 中，因此不需要指定深度编号。提供了用于指定和管理对象的 z 顺序的新方法。

### 处理动态数据和内容

ActionScript 3.0 包含用于加载和处理 Flash 应用程序中的资源和数据的机制，这些机制在 API 中是直观的并且是一致的。新增的 Loader 类提供了一种加载 SWF 文件和图像资源的单一机制，并提供了一种访问已加载内容的详细信息的方法。 URLLoader 类提供了一种单独的机制，用于在数据驱动的应用程序中加载文本和二进制数据。Socket 类提供了一种以任意格式从 / 向服务器套接字中读取 / 写入二进制数据的方式。

### 低级数据访问

各种 API 提供了对数据的低级访问，而这种访问以前在 ActionScript 中是不可能的。对于正在下载的数据而言，可使用 URLStream 类（由 URLLoader 实现）在下载数据的同时访问原始二进制数据。使用 ByteArray 类可优化二进制数据的读取、写入以及处理。使用新增的 Sound API，可以通过 SoundChannel 类和 SoundMixer 类对声音进行精细控制。新增的处理安全性的 API 可提供有关 SWF 文件或加载内容的安全权限的信息，从而使您能够更好地处理安全错误。

### 处理文本

ActionScript 3.0 包含一个用于所有与文本相关的 API 的 flash.text 包。TextLineMetrics 类为文本字段中的一行文本提供精确度量；它取代了 ActionScript 2.0 中的 `TextField.getLineMetrics()` 方法。 TextField 类包含许多有趣的新低级方法，它们可以提供有关文本字段中的一行文本或单个字符的特定信息。这些方法包括 `getCharBoundaries()`（返回一个表示字符边框的矩形）、 `getCharIndexAtPoint()`（返回指定点处字符的索引）以及 `getFirstCharInParagraph()`（返回段落中第一个字符的索引）。行级方法包括 `getLineLength()`（返回指定文本行中的字符数）和 `getLineText()`（返回指定行的文本）。新增的 Font 类提供了一种管理 SWF 文件中的嵌入字体的方法。

## 与早期版本的兼容性

和以往一样， Flash Player 提供针对以前发布的内容的完全向后兼容性。在 Flash Player 9 中，可以运行在早期 Flash Player 版本中运行的任何内容。然而，在 Flash Player 9 中引入 ActionScript 3.0 后，的确对在 Flash Player 9 中运行的旧内容和新内容之间的互操作性提出了挑战。兼容性问题包括以下几个方面：

- 单个 SWF 文件无法将 ActionScript 1.0 或 2.0 代码和 ActionScript 3.0 代码组合在一起。
- ActionScript 3.0 代码可以加载以 ActionScript 1.0 或 2.0 编写的 SWF 文件，但它无法访问该 SWF 文件的变量和函数。
- 以 ActionScript 1.0 或 2.0 编写的 SWF 文件无法加载以 ActionScript 3.0 编写的 SWF 文件。这意味着在 Flash 8 或 Flex Builder 1.5 或更早版本中创作的 SWF 文件无法加载 ActionScript 3.0 SWF 文件。

  此规则的唯一例外情况是，只要 ActionScript 2.0 SWF 文件以前没有向它的任何级别加载任何内容，ActionScript 2.0 SWF 文件就可以用 ActionScript 3.0 SWF 文件来替换它自身。 ActionScript 2.0 SWF 文件可通过调用 `loadMovieNum()` 并将值 0 传递给 `level` 参数来实现此目的。

- 通常，如果以 ActionScript 1.0 或 2.0 编写的 SWF 文件要与以 ActionScript 3.0 编写的 SWF 文件一起工作，则必须进行迁移。例如，假定您使用 ActionScript 2.0 创建了一个媒体播放器。该媒体播放器加载同样是使用 ActionScript 2.0 创建的各种内容。无法将用 ActionScript 3.0 创建的新内容加载到该媒体播放器中。您必须将视频播放器迁移到 ActionScript 3.0。

  但是，如果您在 ActionScript 3.0 中创建一个媒体播放器，则该媒体播放器可以执行 ActionScript 2.0 内容的简单加载。

下表概述了早期的 Flash Player 版本在加载新内容和执行代码方面的局限性，以及在不同的 ActionScript 版本中编写的 SWF 文件之间跨脚本编写的局限性。

| 支持的功能 | 运行时环境 | | |
|---|---|---|---|
| | Flash Player 7 | Flash Player 8 | Flash Player 9 |
| 可以加载针对以下版本发布的 SWF | 7 和更早版本 | 8 和更早版本 | 9 和更早版本 |
| 包含此 AVM | AVM1 | AVM1 | AVM1 和 AVM2 |
| 运行在以下 ActionScript 版本中编写的 SWF | 1.0 和 2.0 | 1.0 和 2.0 | 1.0、2.0 和 3.0 |

| 支持的功能* | 在以下版本中创建的内容 | |
|---|---|---|
| | ActionScript 1.0 和 2.0 | ActionScript 3.0 |
| 可以加载在以下版本中创建的内容并在其中执行代码 | 仅 ActionScript 1.0 和 2.0 | ActionScript 1.0、2.0 和 ActionScript 3.0 |
| 可以对在以下版本中创建的内容进行跨脚本编写 | 仅 ActionScript 1.0 和 2.0† | ActionScript 3.0‡ |

* 运行在 Flash Player 9 或更高版本中的内容。运行在 Flash Player 8 或更早版本中的内容只能在 ActionScript 1.0 和 2.0 中加载、显示、执行以及跨脚本编写。

† ActionScript 3.0 （通过本地连接）。

‡ ActionScript 1.0 和 2.0 （通过本地连接）。

第 2 章

# ActionScript 快速入门

2

本章旨在让您可以着手进行 ActionScript 编程，同时为您提供了解本手册其余部分的概念和示例所需的背景。本章首先讨论基本的编程概念，这些概念在关于如何在 ActionScript 中应用它们的上下文中介绍。本章还介绍了组织和构建 ActionScript 应用程序的要点。

## 目录



# 编程基础

因为 ActionScript 是一种编程语言，所以，如果您首先了解几个通用的计算机编程概念，则会对您学习 ActionScript 很有帮助。

## 计算机程序的用途

首先，对计算机程序的概念及其用途有一个概念性的认识是非常有用的。计算机程序主要包括两个方面：

- 程序是计算机执行的一系列指令或步骤。
- 每一步最终都涉及到对某一段信息或数据的处理。

通常认为，计算机程序只是您提供给计算机并让它逐步执行的指令列表。每个单独的指令都称为“语句”。正如您将在本手册中看到的那样，在 ActionScript 中编写的每个语句的末尾都有一个分号。

实质上，程序中给定指令所做的全部操作就是处理存储在计算机内存中的一些数据位。举一个简单的例子，您可能指示计算机将两个数字相加并将结果存储在计算机的内存中。举一个较复杂的例子：假设在屏幕上绘制了一个矩形，您希望编写一个程序将它移动到屏幕上的其它位置。计算机跟踪该矩形的某些信息 — 该矩形所在位置的 x、y 光标、它的宽度和高度以及颜色等等。这些信息位中的每一位都存储在计算机内存中的某个位置。为了将矩形移动到其它位置，程序将采取类似于 "将 x 坐标改为 200；将 y 坐标改为 150" 的步骤（也就是说，为 x 和 y 坐标指定新值）。当然，计算机的确会对这些数据进行某些处理，以便切实地将这些数字转变为显示在计算机屏幕上的图像；但考虑到我们所感兴趣的详细程度，我们只要知道 "在屏幕上移动矩形" 这一过程确实只涉及更改计算机内存中的数据位就足够了。

## 变量和常量

由于编程主要涉及更改计算机内存中的信息，因此在程序中需要一种方法来表示单条信息。"变量" 是一个名称，它代表计算机内存中的值。在编写语句来处理值时，编写变量名来代替值；只要计算机看到程序中的变量名，就会查看自己的内存并使用在内存中找到的值。例如，如果两个名为 `value1` 和 `value2` 的变量都包含一个数字，您可以编写如下语句以将这两个数字相加：

```
value1 + value2
```

在实际执行这些步骤时，计算机将查看每个变量中的值，并将它们相加。

在 ActionScript 3.0 中，一个变量实际上包含三个不同部分：

- 变量名
- 可以存储在变量中的数据的类型
- 存储在计算机内存中的实际值

刚才我们讨论了计算机是如何将名称用作值的占位符的。数据类型也非常重要。在 ActionScript 中创建变量时，应指定该变量将保存的数据的特定类型；此后，程序的指令只能在该变量中存储此类型的数据，您可以使用与该变量的数据类型关联的特定特性来处理值。在 ActionScript 中，要创建一个变量（称为 "声明" 变量），应使用 `var` 语句：

```
var value1:Number;
```

在本例中，我们指示计算机创建一个名为 `value1` 的变量，该变量仅保存 Number 数据（"Number" 是在 ActionScript 中定义的一种特定数据类型）。您还可以立即在变量中存储一个值：

```
var value2:Number = 17;
```

在 Adobe Flash CS3 Professional 中，还包含另外一种变量声明方法。在将一个影片剪辑元件、按钮元件或文本字段放置在舞台上时，可以在 “属性” 检查器中为它指定一个实例名称。在后台，Flash 将创建一个与该实例名称同名的变量，您可以在 ActionScript 代码中使用该变量来引用该舞台项目。例如，如果您将一个影片剪辑元件放在舞台上并为它指定了实例名称 `rocketShip`，那么，只要您在 ActionScript 代码中使用变量 `rocketShip`，实际上就是在处理该影片剪辑。

## 数据类型

在 ActionScript 中，您可以将很多数据类型用作所创建的变量的数据类型。其中的某些数据类型可以看作是 “简单” 或 “基本” 数据类型：

- String：一个文本值，例如，一个名称或书中某一章的文字
- Numeric：对于 numeric 型数据， ActionScript 3.0 包含三种特定的数据类型：
  - Number：任何数值，包括有小数部分或没有小数部分的值
  - Int：一个整数 （不带小数部分的整数）
  - Uint：一个 “无符号” 整数，即不能为负数的整数
- Boolean：一个 true 或 false 值，例如开关是否开启或两个值是否相等

简单数据类型表示单条信息：例如，单个数字或单个文本序列。然而，ActionScript 中定义的大部分数据类型都可以被描述为复杂数据类型，因为它们表示组合在一起的一组值。例如，数据类型为 Date 的变量表示单个值 — 时间中的某个片刻。然而，该日期值实际上表示为几个值：年、月、日、时、分、秒等等，它们都是单独的数字。所以，虽然我们认为日期是单个值 （可以通过创建一个 Date 变量将日期作为单个值来对待），而在计算机内部却认为日期是组合在一起、共同定义单个日期的一组值。

大部分内置数据类型以及程序员定义的数据类型都是复杂数据类型。您可能认识下面的一些复杂数据类型：

- MovieClip：影片剪辑元件
- TextField：动态文本字段或输入文本字段
- SimpleButton：按钮元件
- Date：有关时间中的某个片刻的信息 （日期和时间）

经常用作数据类型的同义词的两个词是类和对象。“类” 仅仅是数据类型的定义 — 就像用于该数据类型的所有对象的模板，例如 “所有 Example 数据类型的变量都拥有这些特性：A、B 和 C”。而 “对象” 仅仅是类的一个实际的实例；可将一个数据类型为 MovieClip 的变量描述为一个 MovieClip 对象。下面几条陈述虽然表达的方式不同，但意思是相同的：

- 变量 `myVariable` 的数据类型是 Number。
- 变量 `myVariable` 是一个 Number 实例。
- 变量 `myVariable` 是一个 Number 对象。
- 变量 `myVariable` 是 Number 类的一个实例。

# 处理对象

ActionScript 是一种面向对象的编程语言。面向对象的编程仅仅是一种编程方法，它与使用对象来组织程序中的代码的方法没有什么差别。

先前我们将计算机程序定义为计算机执行的一系列步骤或指令。那么从概念上讲，我们可能认为计算机程序只是一个很长的指令列表。然而，在面向对象的编程中，程序指令被划分到不同的对象中 — 代码构成功能块，因此相关类型的功能或相关的信息被组合到一个容器中。

事实上，如果您已经在 Flash 中处理过元件，那么您应已习惯于处理对象了。假设您已定义了一个影片剪辑元件 （假设它是一幅矩形的图画），并且已将它的一个副本放在了舞台上。从严格意义上来说，该影片剪辑元件也是 ActionScript 中的一个对象，即 MovieClip 类的一个实例。

您可以修改该影片剪辑的不同特征。例如，当选中该影片剪辑时，您可以在 "属性"检查器中更改许多值，例如，它的 x 坐标、宽度，进行各种颜色调整 （例如，更改它的 alpha 值，即透明度），或对它应用投影滤镜。还可以使用其它 Flash 工具进行更多更改，例如，使用"任意变形"工具旋转该矩形。在 Flash 创作环境中修改一个影片剪辑元件时所做的更改，同样可在 ActionScript 中通过更改组合在一起、构成称为 MovieClip 对象的单个包的各数据片断来实现。

在 ActionScript 面向对象的编程中，任何类都可以包含三种类型的特性：

- 属性
- 方法
- 事件

这些元素共同用于管理程序使用的数据块，并用于确定执行哪些动作以及动作的执行顺序。

## 属性

属性表示某个对象中绑定在一起的若干数据块中的一个。 Song 对象可能具有名为 `artist` 和 `title` 的属性； MovieClip 类具有 `rotation`、`x`、`width` 和 `alpha` 等属性。您可以像处理单个变量那样处理属性；事实上，可以将属性视为包含在对象中的 "子"变量。

以下是一些使用属性的 ActionScript 代码的示例。以下代码行将名为 `square` 的 MovieClip 移动到 100 个像素的 x 坐标处：

```
square.x = 100;
```

以下代码使用 rotation 属性旋转 `square` MovieClip 以便与 `triangle` MovieClip 的旋转相匹配：

```
square.rotation = triangle.rotation;
```

以下代码更改 `square` MovieClip 的水平缩放比例，以使其宽度为原始宽度的 1.5 倍：

```
square.scaleX = 1.5;
```

请注意上面几个示例的通用结构：将变量（`square` 和 `triangle`）用作对象的名称，后跟一个句点 (.) 和属性名（`x`、`rotation` 和 `scaleX`）。句点称为"点运算符"，用于指示您要访问对象的某个子元素。整个结构"变量名 - 点 - 属性名"的使用类似于单个变量，变量是计算机内存中的单个值的名称。

## 方法

"方法"是指可以由对象执行的操作。例如，如果在 Flash 中使用时间轴上的几个关键帧和动画制作了一个影片剪辑元件，则可以播放或停止该影片剪辑，或者指示它将播放头移到特定的帧。

下面的代码指示名为 `shortFilm` 的 MovieClip 开始播放：

```
shortFilm.play();
```

下面的代码行使名为 `shortFilm` 的 MovieClip 停止播放（播放头停在原地，就像暂停播放视频一样）：

```
shortFilm.stop();
```

下面的代码使名为 `shortFilm` 的 MovieClip 将其播放头移到第 1 帧，然后停止播放（就像后退视频一样）：

```
shortFilm.gotoAndStop(1);
```

正如您所看到的一样，您可以通过依次写下对象名（变量）、句点、方法名和小括号来访问方法，这与属性类似。小括号是指示要"调用"某个方法（即指示对象执行该动作）的方式。有时，为了传递执行动作所需的额外信息，将值（或变量）放入小括号中。这些值称为方法"参数"。例如，`gotoAndStop()` 方法需要知道应转到哪一帧，所以要求小括号中有一个参数。有些方法（如 `play()` 和 `stop()`）自身的意义已非常明确，因此不需要额外信息。但书写时仍然带有小括号。

与属性（和变量）不同的是，方法不能用作值占位符。然而，一些方法可以执行计算并返回可以像变量一样使用的结果。例如，Number 类的 `toString()` 方法将数值转换为文本表示形式：

```
var numericData:Number = 9;
var textData:String = numericData.toString();
```

例如，如果希望在屏幕上的文本字段中显示 Number 变量的值，应使用 `toString()` 方法。TextField 类的 `text` 属性（表示实际在屏幕上显示的文本内容）被定义为 String，所以它只能包含文本值。下面的一行代码将变量 `numericData` 中的数值转换为文本，然后使这些文本显示在屏幕上名为 `calculatorDisplay` 的 TextField 对象中：

```
calculatorDisplay.text = numericData.toString();
```

# 事件

我们已经介绍了计算机程序就是计算机分步执行的一系列指令。一些简单的计算机程序仅包括计算机执行的几个步骤以及程序的结束点。然而，ActionScript 程序可以保持运行、等待用户输入或等待其它事件发生。事件是确定计算机执行哪些指令以及何时执行的机制。

本质上，“事件”就是所发生的、 ActionScript 能够识别并可响应的事情。许多事件与用户交互有关 — 例如，用户单击按钮，或按键盘上的键 — 但也有其它类型的事件。例如，如果使用 ActionScript 加载外部图像，有一个事件可让您知道图像何时加载完毕。本质上，当 ActionScript 程序正在运行时，Adobe Flash Player 只是坐等某些事情的发生，当这些事情发生时， Flash Player 将运行您为这些事件指定的特定 ActionScript 代码。

## 基本事件处理

指定为响应特定事件而应执行的某些动作的技术称为 “事件处理”。在编写执行事件处理的 ActionScript 代码时，您需要识别三个重要元素：

- 事件源：发生该事件的是哪个对象？例如，哪个按钮会被单击，或哪个 Loader 对象正在加载图像？事件源也称为 “事件目标”，因为 Flash Player 将此对象 （实际在其中发生事件）作为事件的目标。
- 事件：将要发生什么事情，以及您希望响应什么事情？识别事件是非常重要的，因为许多对象都会触发多个事件。
- 响应：当事件发生时，您希望执行哪些步骤？

无论何时编写处理事件的 ActionScript 代码，都会包括这三个元素，并且代码将遵循以下基本结构 （以粗体显示的元素是您将针对具体情况填写的占位符）：

```
function eventResponse(eventObject:EventType):void
{
    // 此处是为响应事件而执行的动作。
}

eventSource.addEventListener(EventType.EVENT_NAME, eventResponse);
```

此代码执行两个操作。首先，定义一个函数，这是指定为响应事件而要执行的动作的方法。接下来，调用源对象的 `addEventListener()` 方法，实际上就是为指定事件 “订阅”该函数，以便当该事件发生时，执行该函数的动作。我们将更为详细地讨论其中每个部分。

“函数”提供一种将若干个动作组合在一起、用类似于快捷名称的单个名称来执行这些动作的方法。函数与方法完全相同，只是不必与特定类关联（事实上，方法可以被定义为与特定类关联的函数）。在创建事件处理函数时，必须选择函数名称 （本例中为 `eventResponse`），还必须指定一个参数 （本例中的名称为 `eventObject`）。指定函数参数类似于声明变量，所以还必须指明参数的数据类型。将为每个事件定义一个 ActionScript 类，并且为函数参数指定的数据类型始终是与要响应的特定事件关联的类。最后，在左大括号与右大括号之间 (`{ ...}`)，编写您希望计算机在事件发生时执行的指令。

一旦编写了事件处理函数，就需要通知事件源对象（发生事件的对象，如按钮）您希望在该事件发生时调用函数。可通过调用该对象的 addEventListener() 方法来实现此目的（所有具有事件的对象都同时具有 addEventListener() 方法）。addEventListener() 方法有两个参数：

- 第一个参数是您希望响应的特定事件的名称。同样，每个事件都与一个特定类关联，而该类将为每个事件预定义一个特殊值；类似于事件自己的唯一名称（应将其用于第一个参数）。
- 第二个参数是事件响应函数的名称。请注意，如果将函数名称作为参数进行传递，则在写入函数名称时不使用括号。

## 了解事件处理过程

下面分步描述了创建事件侦听器时执行的过程。在本例中，您将创建一个侦听器函数，在单击名为 myButton 的对象时将调用该函数。

程序员实际编写的代码如下所示：

```
function eventResponse(event:MouseEvent):void
{
   // 此处是为响应事件而执行的动作。
}

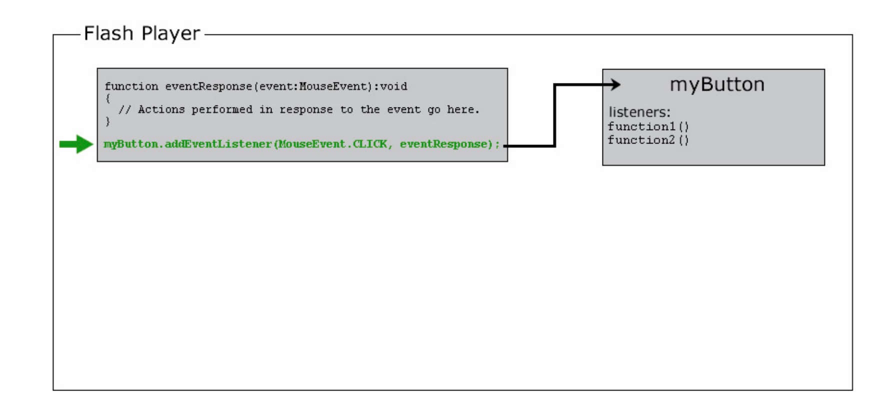
myButton.addEventListener(MouseEvent.CLICK, eventResponse);
```

下面是此代码在 Flash Player 中运行时的实际工作方式：

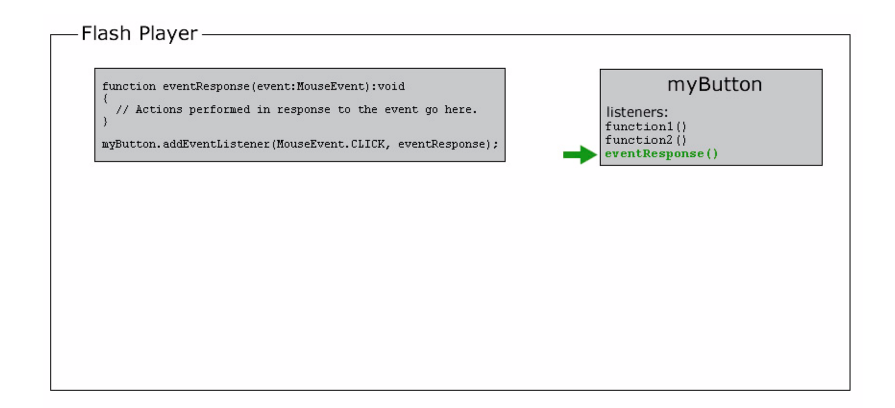
1. 加载 SWF 文件时，Flash Player 会注意到以下情况：有一个名为 eventResponse() 的函数。



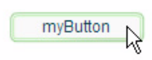
2. Flash Player 随后运行该代码 （具体地说，是指不在函数中的代码行）。在本例中，只有一行代码：针对事件源对象 （名为 myButton）调用 addEventListener() 方法，并将 eventResponse 函数作为参数进行传递。




   a. 在内部，myButton 包含正在侦听其每个事件的函数的列表，因此，当调用其 addEventListener() 方法时，myButton 将 eventResponse() 函数存储在其事件侦听器列表中。

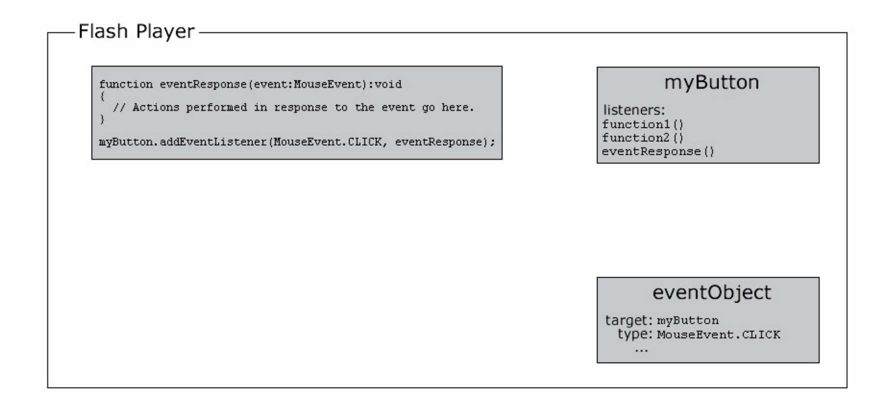


3. 在某一时刻，用户单击 `myButton` 对象以触发其 `click` 事件 （在代码中将其标识为 `MouseEvent.CLICK`）。

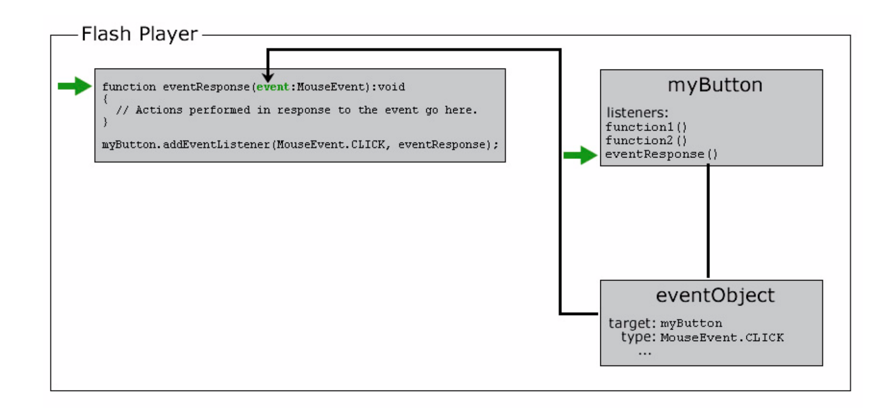



此时发生了以下事件：

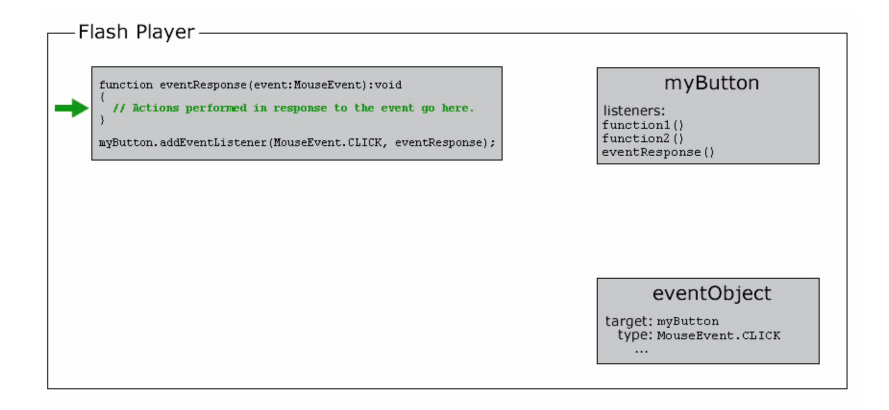
a. Flash Player 创建一个对象，它是与所述事件（本示例中为 MouseEvent）关联的类的实例。对于很多事件，这是 Event 类的实例；对于鼠标事件，这是 MouseEvent 实例；对于其它事件，这是与该事件关联的类的实例。创建的该对象称为 “事件对象”，它包含有关所发生的事件的特定信息：事件类型、发生位置以及其它特定于事件的信息 （如果适用）。



b. Flash Player 随后查看 myButton 存储的事件侦听器的列表。它逐个查看这些函数，以调用每个函数并将事件对象作为参数传递给该函数。由于 eventResponse() 函数是 myButton 的侦听器之一，因此， Flash Player 将在此过程中调用 eventResponse() 函数。




c. 当调用 eventResponse() 函数时，将运行该函数中的代码，因此，将执行您指定的动作。



## 事件处理示例

以下是几个更为具体的事件示例，让您对在编写事件处理代码时常用的一些事件元素以及可能的变化形式有一个了解：

- 单击按钮开始播放当前的影片剪辑。在下面的示例中，playButton 是按钮的实例名称，而 this 是表示 “当前对象”的特殊名称：

```
this.stop();

function playMovie(event:MouseEvent):void
{
  this.play();
}

playButton.addEventListener(MouseEvent.CLICK, playMovie);
```

- 检测文本字段中的键入操作。在下面的示例中，entryText 是一个输入文本字段，而 outputText 是一个动态文本字段：

```
function updateOutput(event:TextEvent):void
{
  var pressedKey:String = event.text;
  outputText.text = "You typed: " + pressedKey;
}

entryText.addEventListener(TextEvent.TEXT_INPUT, updateOutput);
```

- 单击按钮导航到一个 URL。在本例中，linkButton 是该按钮的实例名称：

```
function gotoAdobeSite(event:MouseEvent):void
{
  var adobeURL:URLRequest = new URLRequest("http://www.adobe.com/");
  navigateToURL(adobeURL);
}

linkButton.addEventListener(MouseEvent.CLICK, gotoAdobeSite);
```

# 创建对象实例

当然，在 ActionScript 中使用对象之前，该对象首先必须存在。创建对象的步骤之一是声明变量；然而，声明变量仅仅是在计算机的内存中创建一个空位置。您必须为变量指定实际值 — 即创建一个对象并将它存储在该变量中 — 然后再尝试使用或处理该变量。创建对象的过程称为对象 “实例化”；也就是说，创建特定类的实例。

有一种创建对象实例的简单方法完全不必涉及 ActionScript。在 Flash 中，当将一个影片剪辑元件、按钮元件或文本字段放置在舞台上，并在“属性”检查器中为它指定实例名时，Flash 会自动声明一个拥有该实例名的变量、创建一个对象实例并将该对象存储在该变量中。同样，在 Adobe Flex Builder 中，当您以 Adobe Macromedia® MXML™ 创建一个组件（通过用 MXML 标签进行编码或通过将组件放置在处于设计模式下的编辑器中）并为该组件分配一个 ID（在 MXML 标记中或在 Flex 属性视图中）时，该 ID 将成为一个 ActionScript 变量的名称，并且会创建该组件的一个实例并将它存储在该变量中。

然而，您不会总是希望直观地创建对象。还可以通过几种方法来仅使用 ActionScript 创建对象实例。首先，借助几个 ActionScript 数据类型，可以使用“文本表达式”（直接写入 ActionScript 代码的值）创建一个实例。下面给出了一些示例：

- 文本数字值（直接输入数字）：

```
var someNumber:Number = 17.239;
var someNegativeInteger:int = -53;
var someUint:uint = 22;
```

- 文本字符串值（用双引号将本文引起来）：

```
var firstName:String = "George";
var soliloquy:String = "To be or not to be, that is the question...";
```

- 文本布尔值（使用字面值 `true` 或 `false`）：

```
var niceWeather:Boolean = true;
var playingOutside:Boolean = false;
```

- 文本 XML 值（直接输入 XML）：

```
var employee:XML = <employee>
     <firstName>Harold</firstName>
     <lastName>Webster</lastName>
  </employee>;
```

ActionScript 还为 Array、RegExp、Object 和 Function 数据类型定义了文本表达式。有关这些类的详细信息，请参阅第 189 页的“处理数组”、第 243 页的“使用正则表达式”和第 80 页的“Object 数据类型”。

对于其它任何数据类型而言，要创建一个对象实例，应将 `new` 运算符与类名一起使用，如下所示：

```
var raceCar:MovieClip = new MovieClip();
var birthday:Date = new Date(2006, 7, 9);
```

通常，将使用 new 运算符创建对象称为 “调用类的构造函数”。“构造函数”是一种特殊方法，在创建类实例的过程中将调用该方法。请注意，当以此方法创建实例时，请在类名后加上小括号，有时还可以指定参数值 — 这是在调用方法时另外可执行的两个操作。

**注意**

甚至对于可使用文本表达式创建实例的数据类型，也可以使用 new 运算符来创建对象实例。例如，下面的两行代码执行的是相同的操作：

```
var someNumber:Number = 6.33;
var someNumber:Number = new Number(6.33);
```

熟悉使用 new *ClassName*() 创建对象的方法是非常重要的。如果需要创建无可视化表示形式的 ActionScript 数据类型的一个实例（无法通过将项目放置在 Flash 舞台上来创建，也无法在 Flex Builder MXML 编辑器的设计模式下创建），则只能通过使用 new 运算符在 ActionScript 中直接创建对象来实现此目的。

具体到 Flash 中，new 运算符还可用于创建已在库中定义、但没有放在舞台上的影片剪辑元件的实例。有关详细信息，请参阅第 430 页的 “使用 ActionScript 创建 MovieClip 对象”。

# 常用编程元素

除了声明变量、创建对象实例以及使用属性和方法来处理对象之外，还可以使用其它几个构造块来创建 ActionScript 程序。

## 运算符

“运算符”是用于执行计算的特殊符号（有时候是词）。这些运算符主要用于数学运算，有时也用于值的比较。通常，运算符使用一个或多个值并 “算出” 一个结果。例如：

- 加法运算符 (+) 将两个值相加，结果是一个数字：

  ```
  var sum:Number = 23 + 32;
  ```

- 乘法运算符 (*) 将一个值与另一个值相乘，结果是一个数字：

  ```
  var energy:Number = mass * speedOfLight * speedOfLight;
  ```

- 等于运算符 (==) 比较两个值，看它们是否相等，结果是一个 true 或 false（布尔）值：

  ```
  if (dayOfWeek == "Wednesday")
  {
    takeOutTrash();
  }
  ```

  如上所示，等于运算符和其它 “比较” 运算符通常用于 if 语句，以确定是否应执行某些指令。

  有关使用运算符的更多详细信息和示例，请参阅第 90 页的 “运算符”。

## 注释

在编写 ActionScript 时，您通常会希望给自己留一些注释，这些注释可能解释某些代码行如何工作或者为什么做出特定的选择。“代码注释”是一个工具，用于编写计算机应在代码中忽略的文本。 ActionScript 包括两种注释：

- 单行注释：在一行中的任意位置放置两个斜杠来指定单行注释。计算机将忽略斜杠后直到该行末尾的所有内容：

```
// 这是注释；计算机将会忽略它。
var age:Number = 10; // 默认情况下，将 age 设置为 10。
```

- 多行注释：多行注释包括一个开始注释标记 (/*)、注释内容和一个结束注释标记 (*/)。无论注释跨多少行，计算机都将忽略开始标记与结束标记之间的所有内容：

```
/*
这可能是一段非常长的说明，可能说明
特定函数的作用或解释某一部分代码。

在任何情况下，计算机都将忽略所有这些行。
*/
```

注释的另一种常见用法是临时禁用一行或多行代码 — 例如，如果您要测试执行某操作的其它方法，或要查明为什么某些 ActionScript 代码没有按您期望的方式工作。

## 流控制

在程序中，经常需要重复某些动作，仅执行某些动作而不执行其它动作，或根据某些条件执行替代动作等等。“流控制”就是用于控制执行哪些动作。 ActionScript 中提供了几种类型的流控制元素。

- 函数：函数类似于快捷方式，提供了一种将一系列动作组合到单个名称下的方法，并可用于执行计算。函数对于处理事件尤为重要，但也可用作组合一系列指令的通用工具。有关函数的详细信息，请参阅第 102 页的 “函数”。
- 循环：使用循环结构，可指定计算机反复执行一组指令，直到达到设定的次数或某些条件改变为止。通常借助循环并使用一个其值在计算机每执行完一次循环后就改变的变量来处理几个相关项。有关循环的详细信息，请参阅第 100 页的 “循环”。
- 条件语句：条件语句提供一种方法，用于指定仅在某些情况下才执行的某些指令或针对不同的条件提供不同的指令集。最常见的一类条件语句是 `if` 语句。 `if` 语句检查该语句括号中的值或表达式。如果值为 `true`，则执行大括号中的代码行；否则，将忽略它们。例如：

```
if (age < 20)
{
    // 显示针对青少年的特殊内容
}
```

else 语句与 if 语句一起使用，用于指定在条件不为 true 时执行的替代指令：

```
if (username == "admin")
{
   // 执行一些仅限管理员完成的操作，如显示额外选项
}
else
{
   // 执行一些非管理员完成的操作
}
```

有关条件语句的详细信息，请参阅第 97 页的 “条件语句”。

# 示例：动画公文包片段

该示例的设计目的是让您在第一时机看到如何可以将各段 ActionScript 合并为一个完整的应用程序（如果对于 ActionScript 来说并不过于庞大）。该动画公文包片段是一个示例，演示如何利用现有的线性动画（例如，为客户创建的片段），并添加一些适用于将动画融入在线公文包中的微小的交互式元素。要添加到动画中的交互行为将包括两个用户可以单击的按钮：一个用于启动动画，另一个用于导航到单独的 URL（例如公文包菜单或创作者的主页）。

创建该片段的过程可以分为四个主要部分：

1. 准备 FLA 文件以便添加 ActionScript 和交互式元素。
2. 创建和添加按钮。
3. 编写 ActionScript 代码。
4. 测试应用程序。

## 准备添加交互

向动画添加交互式元素之前，创建一些用于添加新内容的位置将有助于创建 FLA 文件。这包括在舞台上创建可以在其中放置按钮的实际空间，还包括在 FLA 文件中创建 “空间”以使不同的项目保持分开。

**要创建 FLA 以添加交互式元素，请执行下列操作：**

1. 如果您还没有要向其中添加交互的线性动画，请创建一个具有简单动画（例如一个补间动画或补间形状）的新的 FLA 文件。否则，请打开包含您要在本项目中展示的动画的 FLA 文件，并用新名称保存该文件，以创建一个新的工作文件。
2. 决定屏幕上您希望两个按钮（一个用于启动动画，另一个用于链接到创造者公文包或主页）显示的位置。如果需要，在舞台上为该新内容清除或添加一些空间。如果该动画还没有启动画面，您可能需要在第 1 帧中创建一个（您可能需要移动动画，使它在第 2 帧或更后的帧中启动）。

3. 在时间轴中的其它图层上创建一个新图层，并将其重命名为 **buttons**。这将是要在其中添加按钮的图层。
4. 在 buttons 图层之上创建一个新图层，并将其命名为 **actions**。这将是要在其中向应用程序添加 ActionScript 代码的图层。

## 创建和添加按钮

接下来，将需要真正创建和摆放将构成交互式应用程序中心的按钮。

**要创建按钮并将其添加到 FLA，请执行下列操作：**

1. 使用绘图工具在 buttons 图层上创建第一个按钮 （"play"按钮）的可视外观，例如，您可以绘制一个水平椭圆，且椭圆顶部有一些文本。
2. 使用 "选择"工具选择单个按钮的所有图形部分。
3. 在主菜单中，选择 "修改" > "转换为元件"。
4. 在对话框中，选择 "按钮"作为元件类型，为该元件赋予一个名称，然后单击 "确定"。
5. 选择按钮之后，在 "属性"检查器中为按钮赋予实例名称 `playButton`。
6. 重复步骤 1 至 5，创建将指引查看者到达创作者主页的按钮。将该按钮命名为 `homeButton`。

## 编写代码

虽然该应用程序的 ActionScript 代码都是在同一个位置输入的，但是该代码可分为三组功能。该代码需要执行的三个任务为：

- 一旦 SWF 文件开始加载 （当播放头进入第 1 帧时），就停止播放头。
- 侦听一个事件，该事件在用户单击播放按钮时开始播放 SWF 文件。
- 侦听一个事件，该事件在用户单击创作者主页按钮时将浏览器定向至相应的 URL。

**要创建代码，使得在播放头进入第 1 帧时停止播放头，请执行下列操作：**

1. 在 actions 图层的第 1 帧上选择关键帧。
2. 要打开 "动作"面板，请从主菜单中选择 "窗口" > "动作"。
3. 在 "脚本"窗格中，输入以下代码：

```
stop();
```

**要编写代码，使得单击播放按钮时启动动画，请执行下列操作：**

1. 在前面步骤中输入的代码的末尾添加两个空行。
2. 在脚本底部输入以下代码：

```
function startMovie(event:MouseEvent):void
{
  this.play();
}
```

   该代码定义一个名为 `startMovie()` 的函数。调用 `startMovie()` 时，该函数会导致主时间轴开始播放。

3. 在上一步中添加的代码的下一行中，输入以下代码行：

```
playButton.addEventListener(MouseEvent.CLICK, startMovie);
```

   该代码行将 `startMovie()` 函数注册为 `playButton` 的 `click` 事件的侦听器。也就是说，它使得只要单击名为 `playButton` 的按钮，就会调用 `startMovie()` 函数。

**要编写代码，使得单击主页按钮时将浏览器定向至某一个 URL，请执行下列操作：**

1. 在前面步骤中输入的代码的末尾添加两个空行。
2. 在脚本底部输入以下代码：

```
function gotoAuthorPage(event:MouseEvent):void
{
  var targetURL:URLRequest = new URLRequest("http://example.com/");
  navigateToURL(targetURL);
}
```

   该代码定义一个名为 `gotoAuthorPage()` 的函数。该函数首先创建一个代表 URL http://example.com/ 的 URLRequest 实例，然后将该 URL 传递给 `navigateToURL()` 函数，使用户浏览器打开该 URL。

3. 在上一步中添加的代码的下一行中，输入以下代码行：

```
homeButton.addEventListener(MouseEvent.CLICK, gotoAuthorPage);
```

   该代码行将 `gotoAuthorPage()` 函数注册为 `homeButton` 的 `click` 事件的侦听器。也就是说，它使得只要单击名为 `homeButton` 的按钮，就会调用 `gotoAuthorPage()` 函数。

## 测试应用程序

此时，该应用程序应完全可以工作了。让我们测试一下是否如此。

**要测试该应用程序，请执行下列操作：**

1. 从主菜单中，选择“控制” > “测试影片”。Flash 将创建 SWF 文件，并在 Flash Player 窗口中打开该文件。
2. 试用这两个按钮，确保它们按您预期的方式工作。

3. 如果按钮不起作用，请检查下列事项：
   - 这两个按钮是否具有不同的实例名？
   - `addEventListener()` 方法调用使用的名称是否与按钮的实例名相同？
   - `addEventListener()` 方法调用中使用的事件名称是否正确？
   - 为各个函数指定的参数是否正确？（两个函数都应具有一个数据类型为 MouseEvent 的参数。）

   所有这些错误和大多数其它可能的错误都应在您选择 “测试影片” 命令或单击按钮时给出错误消息。在 “编译器错误” 面板中查看编译器错误（这些错误在您第一次选择 “测试影片” 时发生），并在 “输出” 面板中查看运行时错误（当 SWF 正在播放时，单击按钮等操作会导致发生这些错误）。

# 使用 ActionScript 构建应用程序

要编写 ActionScript 来构建应用程序，仅仅了解使用的语法和类名称是远远不够的。虽然本手册中的大部分信息都围绕这两个主题（语法和使用 ActionScript 类），但是您还希望了解其它一些信息，例如，哪些程序可用于编写 ActionScript，如何组织 ActionScript 代码并将其包括在应用程序中，以及在开发 ActionScript 应用程序时应遵循哪些步骤。

## 用于组织代码的选项

您可以使用 ActionScript 3.0 代码来实现任何目的，从简单的图形动画到复杂的客户端 — 服务器事务处理系统都可以通过它来实现。您可能希望使用一种或多种不同的方法在项目中包括 ActionScript，具体取决于要构建的应用程序类型。

### 将代码存储在 Flash 时间轴中的帧中

在 Flash 创作环境中，可以向时间轴中的任何帧添加 ActionScript 代码。该代码将在影片播放期间播放头进入该帧时执行。

通过将 ActionScript 代码放在帧中，可以方便地向使用 Flash 创作工具构建的应用程序添加行为。您可以将代码添加到主时间轴中的任何帧，或任何 MovieClip 元件的时间轴中的任何帧。但是，这种灵活性也有一定的代价。构建较大的应用程序时，这会容易导致无法跟踪哪些帧包含哪些脚本。这会使得随着时间的过去，应用程序越来越难以维护。

许多开发人员将代码仅仅放在时间轴的第 1 帧中，或放在 Flash 文档中的特定图层上，以简化在 Flash 创作工具中组织其 ActionScript 代码的工作。这样，就可以容易地在 Flash FLA 文件中查找和维护代码。但是，要在另一个 Flash 项目中使用相同的代码，您必须将代码复制并粘贴到新文件中。

如果想要以后能够在其它 Flash 项目中使用您的 ActionScript 代码，您需要将代码存储在外部 ActionScript 文件（扩展名为 .as 的文本文件）中。

## 将代码存储在 ActionScript 文件中

如果您的项目中包括重要的 ActionScript 代码，则最好在单独的 ActionScript 源文件（扩展名为 .as 的文本文件）中组织这些代码。可以采用以下两种方式之一来设置 ActionScript 文件的结构，具体取决于您打算如何在应用程序中使用该文件。

- 非结构化 ActionScript 代码：编写 ActionScript 代码行（包括语句或函数定义），就好像它们是直接在时间轴脚本、 MXML 文件等文件中输入的一样。

  使用 ActionScript 中的 `include` 语句或 Adobe Flex MXML 中的 `<mx:Script>` 标签，可以访问以此方式编写的 ActionScript。ActionScript `include` 语句会导致在特定位置以及脚本中的指定范围内插入外部 ActionScript 文件的内容，就好像它们是直接在那里输入的一样。在 Flex MXML 语言中，可使用 `<mx:Script>` 标签来指定源属性，从而标识要在应用程序中的该点处加载的外部 ActionScript 文件。例如，以下标签将加载名为 Box.as 的外部 ActionScript 文件：

  ```
  <mx:Script source="Box.as" />
  ```

- ActionScript 类定义：定义一个 ActionScript 类，包含它的方法和属性。

  定义一个类后，您就可以像对任何内置的 ActionScript 类所做的那样，通过创建该类的一个实例并使用它的属性、方法和事件来访问该类中的 ActionScript 代码。这要求做到下面两点：

  - 使用 `import` 语句来指定该类的全名，以便 ActionScript 编译器知道可以在哪里找到它。例如，如果您希望使用 ActionScript 中的 MovieClip 类，首先需要使用其全名（包括包和类）来导入该类：

    ```
    import flash.display.MovieClip;
    ```

    或者，您也可以导入包含该 MovieClip 类的包，这与针对该包中的每个类编写单独的 `import` 语句是等效的：

    ```
    import flash.display.*;
    ```

    此规则的唯一例外是，如果在代码中引用的类为顶级类，没有在包中定义，则必须导入该类。

    在 Flash 中，将自动为附加到时间轴上的帧的脚本导入内置类（在 flash.* 包中）。但是，如果您编写自己的类、处理 Flash 创作组件（fl.* 包）或在 Flex 中工作，则需要显式地导入任何类以编写用于创建该类实例的代码。

  - 编写明确引用类名的代码（通常声明一个用该类作为其数据类型的变量，并创建该类的一个实例以便存储在该变量中）。在 ActionScript 代码中引用其它类名即通知编译器加载该类的定义。例如，如果给定一个称为 Box 的外部类，下面的语句将导致创建 Box 类的一个新实例：

    ```
    var smallBox:Box = new Box(10,20);
    ```

    当编译器第一次遇到对 Box 类的引用时，它会搜索加载的源代码以找到 Box 类的定义。

## 选择合适的工具

根据项目需求和可用资源，您可能希望使用几个工具中的一个（或结合使用多个工具）来编写和编辑 ActionScript 代码。

### Flash 创作工具

除了创建图形和动画的功能之外，Adobe Flash CS3 Professional 还包括处理 ActionScript 代码（附加到 FLA 文件中的元素的代码，或仅包含 ActionScript 代码的外部文件中的代码）的工具。 Flash 创作工具最适合于涉及大量的动画或视频的项目，或者您希望自己创建大部分图形资源的项目，尤其适合于用户交互很少或者具有需要 ActionScript 的功能的项目。如果您希望在同一个应用程序中既创建可视资源又编写代码，也可能会选择使用 Flash 创作工具来开发 ActionScript 项目。如果您希望使用预置的用户界面组件，但 SWF 较小或便于设置可视外观是项目的主要考虑因素，那么您也可能会选择使用 Flash 创作工具。

Adobe Flash CS3 Professional 包括两个编写 ActionScript 代码的工具：

- “动作”面板：在 FLA 文件中工作时可用，该面板允许您编写附加到时间轴上的帧的 ActionScript 代码。
- “脚本”窗口：“脚本”窗口是专门用于处理 ActionScript (.as) 代码文件的文本编辑器。

### Flex Builder

Adobe Flex Builder 是创建带有 Flex 框架的项目的首选工具。除了可视布局和 MXML 编辑工具之外， Flex Builder 还包括一个功能完备的 ActionScript 编辑器，因此可用于创建 Flex 或仅包含 ActionScript 的项目。 Flex 应用程序具有以下几个优点：包含一组内容丰富的预置用户界面控件和灵活的动态布局控件，内置了用于处理外部数据源的机制，以及将外部数据链接到用户界面元素。但由于需要额外的代码来提供这些功能，因此 Flex 应用程序的 SWF 文件可能比较大，并且无法像 Flash 应用程序那样轻松地完全重设外观。

如果希望使用 Flex 创建功能完善、数据驱动且内容丰富的 Internet 应用程序，并在一个工具内编辑 ActionScript 代码，编辑 MXML 代码，直观地设置应用程序布局，则应使用 Flex Builder。

## 第三方 ActionScript 编辑器

由于 ActionScript (.as) 文件存储为简单的文本文件，因此任何能够编辑纯文本文件的程序都可以用来编写 ActionScript 文件。除了 Adobe 的 ActionScript 产品之外，还有几个拥有特定于 ActionScript 的功能的第三方文本编辑程序。您可以使用任何文本编辑程序来编写 MXML 文件或 ActionScript 类。然后，可以使用 Flex SDK（包括 Flex 框架类和 Flex 编译器）来基于这些文件创建 SWF 应用程序 （Flex 或仅包含 ActionScript 的应用程序） 。或者，很多开发人员也可以使用第三方 ActionScript 编辑器来编写 ActionScript 类，并结合使用 Flash 创作工具来创建图形内容。

在以下情况下，您可以选择使用第三方 ActionScript 编辑器：

- 您希望在单独的程序中编写 ActionScript 代码，而在 Flash 中设计可视元素。
- 将某个应用程序用于非 ActionScript 编程（例如，创建 HTML 页或以其它编程语言构建应用程序），并希望将该应用程序也用于 ActionScript 编码。
- 您希望使用 Flex SDK 而不用 Flash 和 Flex Builder 来创建仅包含 ActionScript 的项目或 Flex 项目。

有一些提供特定于 ActionScript 的支持的代码编辑器值得注意，其中包括：

- Adobe Dreamweaver® CS3
- ASDT
- FDT
- FlashDevelop
- PrimalScript
- SE|PY
- Xcode （带有 ActionScript 模板和代码提示文件）

# ActionScript 开发过程

无论 ActionScript 项目是大还是小，遵循一个过程来设计和开发应用程序都有助于您提高工作效率。下面几个步骤说明了构建使用 ActionScript 3.0 的应用程序的基本开发过程：

1. 设计应用程序。

   您应先以某种方式描述应用程序，然后再开始构建该应用程序。

2. 编写 ActionScript 3.0 代码。

   您可以使用 Flash、Flex Builder、Dreamweaver 或文本编辑器来创建 ActionScript 代码。

3. 创建 Flash 或 Flex 应用程序文件来运行代码。

   在 Flash 创作工具中，此步骤包括：创建新的 FLA 文件、设置发布设置、向应用程序添加用户界面组件以及引用 ActionScript 代码。在 Flex 开发环境中，创建新的应用程序文件涉及：定义该应用程序并使用 MXML 来添加用户界面组件以及引用 ActionScript 代码。

4. 发布和测试 ActionScript 应用程序。

   这涉及在 Flash 创作环境或 Flex 开发环境中运行应用程序，确保该应用程序执行您期望的所有操作。

请注意：不必按顺序执行这些步骤，或者说不必在完全完成一个步骤后再执行另一步骤。例如，您可能先设计应用程序的一个屏幕 （步骤 1），然后创建图形、按钮等等 （步骤 3），最后再编写 ActionScript 代码（步骤 2）并进行测试（步骤 4）。您也可能先设计应用程序的一部分，然后再一次添加一个按钮或一个界面元素，并为每个按钮或界面元素编写 ActionScript，并在生成后对它进行测试。虽然记住开发过程的这 4 个阶段是十分有用的，但在实际的开发过程中适当地调整各个阶段的顺序通常有助于提高效率。

# 创建自己的类

创建在项目中使用的类的过程可能令人望而生畏。但是，此过程中更难的部分是设计类，即确定类中将包含的方法、属性和事件。

## 类设计策略

面向对象的设计这一主题较为复杂；整个行业的人员都对此学科进行了大量的学术研究和专业实践。尽管如此，下面还是给出了几条建议以帮助您着手进行面向对象的编程。

1. 请考虑一下该类的实例将在应用程序中扮演的角色。通常，对象担任以下三种角色之一：
   - 值对象：这些对象主要用作数据的容器 — 也就是说，它们可能拥有若干个属性和很少的几个方法（有时没有方法）。值对象通常是明确定义的项目的代码表示，例如音乐播放器应用程序中的 Song 类 （表示单个实际的歌曲）或 Playlist 类 （表示概念上的一组歌曲）。
   - 显示对象：它们是实际显示在屏幕上的对象。例如，用户界面元素（如下拉列表或状态显示）和图形元素 （如视频游戏中的角色）等等就是显示对象。
   - 应用程序结构：这些对象在应用程序执行的逻辑或处理方面扮演着广泛的支持角色。例如，在仿生学中执行某些计算的对象；在音乐播放器应用程序中负责刻度盘控件与音量显示之间的值同步的对象；管理视频游戏中的规则的对象；或者在绘画应用程序中加载保存的图片的对象。
2. 确定类所需的特定功能。不同类型的功能通常会成为类的方法。
3. 如果打算将类用作值对象，请确定实例将要包含的数据。这些项是很好的候选属性。

4. 由于类是专门为项目而设计的，因此最重要的是提供应用程序所需的功能。回答下列问题可能会对您很有帮助：
   - 应用程序将存储、跟踪和处理哪些信息？确定这些信息有助于您识别可能需要的值对象和属性。
   - 需要执行哪些操作 — 例如，在应用程序首次加载时，在单击特定的按钮时，在影片停止播放时，分别需要执行哪些操作？这些是很好的候选方法 （如果 "动作" 仅涉及更改单个值，则是很好的候选属性）。
   - 对于任何给定的动作，要执行该动作，该类需要了解哪些信息？这些信息将成为方法的参数。
   - 随着应用程序开始工作，应用程序的其它部分需要了解类中的哪些内容将发生更改？这些是很好的候选事件。
5. 如果有一个现有的对象与您需要的对象类似，只是缺少某些您需要添加的一些额外功能，应考虑创建一个子类 （在现有类的功能的基础之上构建的类，不需要定义它自己的所有功能）。例如，如果您希望创建一个将作为屏幕上的可视对象的类，可将一个现有显示对象（如 Sprite 或 MovieClip）的行为用作该类的基础。在这种情况下，MovieClip（或 Sprite）是 "基类"，而您的类是该类的扩展。有关创建子类的详细信息，请参阅第 134 页的 "继承"。

## 编写类的代码

一旦制订了类的设计计划，或至少对该类需要跟踪哪些信息以及该类需要执行哪些动作有了一定了解后，编写类的实际语法就变得非常简单了。

下面是创建自己的 ActionScript 类的最基本步骤：

1. 在特定于 ActionScript 的程序 （如 Flex Builder 或 Flash）、通用编程工具 （如 Dreamweaver）或者可用来处理纯文本文档的任何程序中打开一个新的文本文档。
2. 输入 `class` 语句定义类的名称。为此，输入单词 `public class`，然后输入类名，后跟一个左大括号和一个右大括号，两个括号之间将是类的内容 （方法和属性定义）。例如：

   ```
   public class MyClass
   {
   }
   ```

   单词 `public` 表示可以从任何其它代码中访问该类。有关其它替代方法，请参阅第 120 页的 "访问控制命名空间属性"。

3. 键入 package 语句以指示包含该类的包的名称。语法是单词 package，后跟完整的包名称，再跟左大括号和右大括号（括号之间将是 class 语句块）。例如，我们将上一步中的代码改为：

```
package mypackage
{
    public class MyClass
    {
    }
}
```

4. 使用 var 语句，在类体内定义该类中的每个属性；语法与用于声明任何变量的语法相同（并增加了 public 修饰符）。例如，在类定义的左大括号与右大括号之间添加下列行将创建名为 textVariable、numericVariable 和 dateVariable 属性：

```
public var textVariable:String = "some default value";
public var numericVariable:Number = 17;
public var dateVariable:Date;
```

5. 使用与函数定义所用的相同语法来定义类中的每个方法。例如：

   - 要创建 myMethod() 方法，应输入：

   ```
   public function myMethod(param1:String, param2:Number):void
   {
       // 使用参数执行某个操作
   }
   ```

   - 要创建一个构造函数（在创建类实例的过程中调用的特殊方法），应创建一个名称与类名称完全匹配的方法：

   ```
   public function MyClass()
   {
       // 为属性设置初始值
       // 否则创建该对象
       textVariable = "Hello there!";
       dateVariable = new Date(2001, 5, 11);
   }
   ```

   如果没有在类中包括构造函数方法，编译器将自动在类中创建一个空构造函数（没有参数和语句）。

您还可以定义其它几个类元素。这些元素更为复杂。

- “存取器”是方法与属性之间的一个特殊交点。在编写代码来定义类时，可以像编写方法一样来编写存取器，这样就可以执行多个动作（而不是像在定义属性时那样，只能读取值或赋值）。但是，在创建类的实例时，可将存取器视为属性 — 仅使用名称来读取值或赋值。有关详细信息，请参阅第 127 页的 “get 和 set 存取器方法”。
- ActionScript 中的事件不是使用特定的语法来定义的。应使用 EventDispatcher 类的功能来定义类中的事件，以便跟踪事件侦听器并将事件通知给它们。有关在您自己的类中创建事件的详细信息，请参阅第 267 页的第 10 章 “处理事件”。

## 有关组织类的一些建议

与早期的 ActionScript 版本不同，ActionScript 3.0 没有一个文件对应一个类的限制，即不限制每个文件只能使用一个类。使用 ActionScript 3.0，您可以将多个类的源代码保存到单个 .as 文件中。在某些情况下，将多个类包装到单个源文件中可能看似比较方便，但通常而言，这被认为是不好的编程习惯，原因有二：

- 如果将多个类包装到一个大文件中，则重用每个类将非常困难。
- 当文件名与类名不对应时，找到特定类的源代码将非常困难。

由于以上原因，Adobe 建议您始终将每个类的源代码保存在其自己的文件中，并为文件指定与类相同的名称。

# 示例：创建基本应用程序

可以使用 Flash、 Flex Builder、 Dreamweaver 或任何文本编辑器来创建一个扩展名为 .as 的外部 ActionScript 源文件。

ActionScript 3.0 可以用在许多应用程序开发环境（包括 Flash 创作工具和 Flex Builder 工具）中。

本节将引导您完成使用 Flash 创作工具或 Flex Builder 2 工具创建和增强一个简单的 ActionScript 3.0 应用程序的步骤。您构建的应用程序将呈现出一种在 Flash 和 Flex 应用程序中使用外部 ActionScript 3.0 类文件的简单模式。在本手册中该模式将会应用于其它所有应用程序范例。

## 设计 ActionScript 应用程序

您应先对要构建的应用程序有一些想法，然后再开始构建该应用程序。

设计的表示可以像应用程序的名称和应用程序用途的简要说明一样简单，也可以像一组包含许多统一建模语言 (UML) 图示的需求文档一样复杂。本手册将不会详细讨论软件设计学科，但希望提醒读者记住应用程序设计是开发 ActionScript 应用程序的一个重要步骤。

我们的第一个 ActionScript 应用程序示例是一个标准的 “Hello World”应用程序，它的设计非常简单：

- 该应用程序将被称为 HelloWorld。
- 它显示包含 “Hello World!”字样的单个文本字段。
- 为了便于重用，该应用程序将使用一个名为 Greeter 的面向对象的类，该类既可以在 Flash 文档中使用，也可以在 Flex 应用程序中使用。
- 在创建该应用程序的一个基本版本之后，您将添加新功能，让用户输入一个用户名并让应用程序对照已知用户列表来检查该用户名。

有了这一简明的定义之后，您可以开始构建该应用程序了。

## 创建 HelloWorld 项目和 Greeter 类

Hello World 应用程序的设计说明指出该应用程序的代码应易于重用。为了实现此目标，该应用程序使用一个名为 Greeter 的面向对象的类，该类可以在 Flex Builder 或 Flash 创作工具创建的应用程序中使用。

**要在 Flash 创作工具中创建 Greeter 类，请执行下列操作：**

1. 在 Flash 创作工具中，选择 “文件” > “新建”。
2. 在 “新建文档”对话框中，选择 “ActionScript 文件”，然后单击 “确定”。

   将显示一个新的 ActionScript 编辑窗口。
3. 选择 “文件” > “保存”。选择一个文件夹以包含您的应用程序，将 ActionScript 文件命名为 **Greeter.as**，然后单击 “确定”。

   继续执行第 48 页的 “在 Greeter 类中添加代码”。

## 在 Greeter 类中添加代码

Greeter 类定义一个对象 `Greeter`，您可以在 HelloWorld 应用程序中使用该对象。

**要向 Greeter 类中添加代码，请执行以下操作：**

1. 将以下代码键入该新文件中：

```
package
{
   public class Greeter
   {
      public function sayHello():String
      {
         var greeting:String;
         greeting = "Hello World!";
         return greeting;
      }
   }
}
```

   Greeter 类包括一个 `sayHello()` 方法，它为指定的用户名返回字符串 “Hello”。
2. 选择 “文件” > “保存”保存此 ActionScript 文件。

现在 Greeter 类可以在 Flash 或 Flex 应用程序中使用了。

## 创建使用 ActionScript 代码的应用程序

您构建的 Greeter 类定义一组自包含的软件功能，但它不表示完整的应用程序。要使用该类，需要创建一个 Flash 文档或 Flex 应用程序。

HelloWorld 应用程序创建 Greeter 类的一个新实例。下面说明了如何将 Greeter 类附加到应用程序。

**要使用 Flash 创作工具创建 ActionScript 应用程序，请执行下列操作：**

1. 选择 "文件" > "新建"。
2. 在 "新建文档" 对话框中，选择 "Flash 文档"，然后单击 "确定"。

   将显示一个新的 Flash 窗口。
3. 选择 "文件" > "保存"。选择包含该 Greeter.as 类文件的同一文件夹，将 Flash 文档命名为 **HelloWorld.fla**，然后单击 "确定"。
4. 在 Flash 的 "工具" 调色板中，选择 "文本" 文件，然后在舞台中拖动以定义一个新的文本字段，该字段宽约 300 像素，高约 100 像素。
5. 在 "属性" 窗口中，保持选中舞台上的文本字段，键入 `mainText` 作为该文本字段的实例名。
6. 单击主时间轴的第 1 帧。
7. 在 "动作" 面板中键入以下脚本：

   ```
   var myGreeter:Greeter = new Greeter();
   mainText.text = myGreeter.sayHello("Bob");
   ```
8. 保存该文件。

继续执行第 49 页的 "发布和测试 ActionScript 应用程序"。

## 发布和测试 ActionScript 应用程序

软件开发是一个重复的过程。编写一些代码，尝试编译这些代码，然后编辑代码，直到能够完全编译这些代码为止。运行编译后的应用程序，测试该应用程序，看它是否实现了预期的设计，如果没有，应再次编辑代码，直到实现预期的设计为止。 Flash 和 Flex Builder 开发环境提供了多种发布、测试和调试应用程序的方法。

下面是在每种环境中测试 HelloWorld 应用程序的基本步骤。

**要使用 Flash 创作工具发布和测试 ActionScript 应用程序，请执行下列操作：**

1. 发布您的应用程序并观察编译错误。在 Flash 创作工具中，选择 "控制" > "测试影片"，编译您的 ActionScript 代码并运行 HelloWorld 应用程序。
2. 如果测试应用程序时 "输出" 窗口中显示任何错误或警告，请在 HelloWorld.fla 或 HelloWorld.as 文件中修复导致这些错误的根源，然后重新测试该应用程序。

3. 如果没有编译错误，您将看到一个显示 Hello World 应用程序的 Flash Player 窗口。其中显示 “Hello, Bob”文本。

您刚才创建了一个简单但完整的面向对象的应用程序，该应用程序使用 ActionScript 3.0。继续执行第 50 页的 “改进 HelloWorld 应用程序”。

## 改进 HelloWorld 应用程序

要使该应用程序更有趣，现在让应用程序要求用户输入用户名并对照预定义的名称列表来验证该用户名。

首先，更新 Greeter 类以添加新功能。然后更新 Flex 或 Flash 应用程序以使用新功能。

**要更新 Greeter.as 文件，请执行以下操作：**

1. 打开 Greeter.as 文件。
2. 将文件的内容更改为如下内容 （新行和更改的行以粗体显示）：

```
package
{
  public class Greeter
  {
    /**
     * Defines the names that should receive a proper greeting.
     */
    public static var validNames:Array = ["Sammy", "Frank", "Dean"];

    /**
     * Builds a greeting string using the given name.
     */
    public function sayHello(userName:String = ""):String
    {
      var greeting:String;
      if (userName == "")
      {
        greeting = "Hello. Please type your user name, and then press the
  Enter key.";
      }
      else if (validName(userName))
      {
        greeting = "Hello, " + userName + ".";
      }
      else
      {
        greeting = "Sorry, " + userName + ", you are not on the list.";
      }
      return greeting;
    }
```

```
    /**
     * Checks whether a name is in the validNames list.
     */
    public static function validName(inputName:String = ""):Boolean
    {
      if (validNames.indexOf(inputName) > -1)
      {
        return true;
      }
      else
      {
        return false;
      }
    }
  }
}
```

现在，Greeter 类拥有许多新功能：

- validNames 数组列出了有效的用户名。在加载 Greeter 类时该数组将初始化为包含三个名称的列表。
- sayHello() 方法现在接受一个用户名并根据一些条件来更改问候语。如果 userName 是一个空字符串 ("")，则 greeting 属性将设置为提示用户输入用户名的语句。如果用户名有效，则问候语为 "Hello, *userName*"。最后，如果这两个条件都不满足，greeting 变量将设置为 "Sorry, *userName*, you are not on the list."。
- 如果在 validNames 数组中找到 inputName，则 validName() 方法将返回 true；否则，返回 false。语句 validNames.indexOf(inputName) 对照 inputName 字符串检查 validNames 数组中的每个字符串。Array.indexOf() 方法返回数组中某个对象的第一个实例的索引位置；如果在数组中找不到该对象，则返回值 -1。

接下来您将编辑引用该 ActionScript 类的 Flash 或 Flex 文件。

**要使用 Flash 创作工具修改该应用程序，请执行下列操作：**

1. 打开 HelloWorld.fla 文件。
2. 修改第 1 帧中的脚本，将一个空字符串 ("") 传递到 Greeter 类的 sayHello() 方法：

```
var myGreeter:Greeter = new Greeter();
mainText.text = myGreeter.sayHello("");
```

3. 在“工具”调色板中选择“文本”工具，然后在舞台上创建两个新的文本字段，这两个字段并排放置在现有的 mainText 文本字段之下。
4. 在第一个新文本字段中，键入文本 **User Name:** 作为标签。
5. 选择另一个新文本字段，并在“属性”检查器中选择 InputText 作为该文本字段的类型。键入 textIn 作为实例名。
6. 单击主时间轴的第 1 帧。

7. 在 “动作” 面板中，在现有脚本的末尾添加以下行：

```
mainText.border = true;
textIn.border = true;

textIn.addEventListener(KeyboardEvent.KEY_UP, keyPressed);

function keyPressed(event:Event):void
{
  if (event.keyCode == Keyboard.ENTER)
  {
    mainText.text = myGreeter.sayHello(textIn.text);
  }
}
```

新代码添加以下功能：

- 前两行只定义两个文本字段的边框。
- 输入文本字段 （例如 `textIn` 字段）具有一组可以调度的事件。使用 `addEventListener()` 方法可以定义一个函数，该函数在发生某一类型的事件时运行。在本例中，该事件指的是按键盘上的 Enter 键。
- `keyPressed()` 自定义函数调用 `myGreeter` 对象的 `sayHello()` 方法，传递 `textIn` 文本字段中的文本作为参数。该方法基于传入的值返回字符串 “greeting”。返回的字符串随后分配给 `mainText` 文本字段的 `text` 属性。

第 1 帧的完整脚本如下所示：

```
mainText.border = true;
textIn.border = true;

var myGreeter:Greeter = new Greeter();
mainText.text = myGreeter.sayHello("");

textIn.addEventListener(KeyboardEvent.KEY_UP, keyPressed);

function keyPressed(event:Event):void
{
  if (event.keyCode == Keyboard.ENTER)
  {
    mainText.text = myGreeter.sayHello(textIn.text);
  }
}
```

8. 保存该文件。

9. 选择 “控制” > “测试影片”，运行应用程序。

   运行该应用程序时，将提示您输入用户名。如果用户名有效 （Sammy、 Frank 或 Dean），该应用程序将显示 “hello” 确认消息。

# 运行后续示例

请注意，您已开发并运行了“Hello World”ActionScript 3.0 应用程序，您应掌握了运行本手册中提供的其它代码示例所需的基础知识。

## 测试本章内的示例代码清单

完成本手册内容的过程中，您可能需要试验用于阐释各个主题的示例代码清单。该测试可能包括显示程序中特定点的变量值，也可能包括查看屏幕上的内容或与该内容交互。为了测试可视内容或交互，将在代码清单之前或代码清单中描述一些必需的元素：您只需要使用所述的元素创建一个文档以测试代码。如果您想要查看程序中某个给定点的变量值，可以通过几种方法来实现。一种方法是使用调试器，例如 Flex Builder 和 Flash 中的内置调试器。但是，对于简单的测试，最简单的方法是只需将变量值输出到某个您可以查看的位置。

下列步骤将帮助您创建一个 Flash 文档，您可以使用该文档来测试代码清单并查看变量值：

**要创建用于测试本章内示例的 Flash 文档，请执行下列操作：**

1. 创建一个新的 Flash 文档，并将其保存在硬盘驱动器上。
2. 要在舞台上的文本字段中显示测试值，请激活 “文本”工具并在舞台上创建一个新的动态文本字段。最有用的是一个较宽、较高的文本字段，该字段的“线条类型”设置为“多行”，且启用了边框。在 “属性”检查器中，为该文本字段赋予一个实例名 （例如，“outputText”）。要将值写入该文本字段中，您需要将调用 `appendText()` 方法的代码添加到示例代码中 （如下所述）。
3. 或者，可以将一个 `trace()` 函数调用添加到代码清单中（如下所述）以查看示例的结果。
4. 要测试给定的示例，请将代码清单复制到“动作”面板中；如果需要，添加一个 `trace()` 函数调用，或使用其 `appendText()` 方法向文本字段添加值。
5. 从主菜单中，选择 “控制” > “测试影片”以创建一个 SWF 文件并查看结果。

既然该方法的作用是查看变量的值，您可以通过两种途径在试验示例时方便地查看变量值：将值写入舞台上的文本字段实例中，或使用 `trace()` 函数将值输出到 “输出”面板。

- `trace()` 函数：ActionScript `trace()` 函数将传递给它的任何参数的值（变量或文本表达式）写入 “输出”面板。本手册中的许多示例列表已经包括了一个 `trace()` 方法调用，因此对于这些列表，您将只需要将代码复制到您的文档中并测试项目。如果要使用 `trace()` 测试不包括该方法的代码清单中的变量值，只需向该代码清单添加一个 `trace()` 调用，传递该变量作为参数。例如，如果遇到类似本章中的代码清单：

```
var albumName:String = "Three for the money";
```

您可以将代码复制到 “动作” 面板，然后添加对 trace() 函数的调用，以测试代码清单的结果：

```
var albumName:String = "Three for the money";
trace("albumName =", albumName);
```

运行该程序时，将输出以下行：

```
albumName = Three for the money
```

每个 trace() 函数调用可以采用多个参数，这些参数排列在一起，组成一个输出行。每个 trace() 函数调用的末尾添加了换行符，因此各个 trace() 调用将在单独的行中输出。

■ 舞台上的文本字段：如果您不想使用 trace() 函数，可以使用 “文本” 工具向舞台添加一个动态文本字段，将值写出到该文本字段中，以查看代码清单的结果。可以使用 TextField 类的 appendText() 方法在该文本字段内容的末尾添加一个字符串值。要使用 ActionScript 访问该文本字段，必须在 “属性” 检查器中为其赋予一个实例名。例如，如果您的文本字段具有实例名 outputText，则可以使用以下代码来查看变量 albumName 的值：

```
var albumName:String = "Three for the money";
outputText.appendText("albumName = ");
outputText.appendText(albumName);
```

该代码将以下文本写入名为 outputText 的文本字段中：

```
albumName = Three for the money
```

如该示例所示，appendText() 方法将文本添加到与前面的内容相同的行中，因此，可以使用多个 appendText() 调用向同一文本行中添加多个值。要强制将文本写入下一行，您可以追加一个换行符 ("\n")：

```
outputText.appendText("\n"); // adds a line break to the text field
```

与 trace() 函数不同， appendText() 方法仅接受一个值作为参数。该值必须是字符串（字符串实例或字符串文本）。要输出非字符串变量的值，必须首先将值转换为字符串。最简单的方法是调用对象的 toString() 方法：

```
var albumYear:int = 1999;
outputText.appendText("albumYear = ");
outputText.appendText(albumYear.toString());
```

## 处理章节结尾的示例

与本章一样，本手册中的大多数章节的结尾都包含一个重要示例，它将该章节中讨论的很多内容串联在一起。但是，与本章中的 Hello World 示例不同，这些示例不是以分步教程的形式提供的。每个示例中相关的 ActionScript 3.0 代码将突出显示并加以讨论，但是不会提供在特定开发环境中运行示例的说明。但是，与本手册一起发布的示例文件将包括在所选择的开发环境中轻松编译和运行示例所需的全部文件。

第 3 章

# ActionScript 语言及其语法

3

ActionScript 3.0 既包含 ActionScript 核心语言又包含 Adobe Flash Player 应用程序编程接口 (API)。 ActionScript 核心语言是 ActionScript 的一部分，实现了 ECMAScript (ECMA-262) 第 4 版语言规范草案。 Flash Player API 提供对 Flash Player 的编程访问。

本章简要介绍 ActionScript 核心语言及其语法。在阅读完本章之后，您应能够对如何处理数据类型和变量、如何使用正确的语法以及如何控制程序中的数据流等方面有一个基本的了解。

## 目录

# 语言概述

对象是 ActionScript 3.0 语言的核心 — 它们是 ActionScript 3.0 语言的基本构造块。您所声明的每个变量、所编写的每个函数以及所创建的每个类实例都是一个对象。可以将 ActionScript 3.0 程序视为一组执行任务、响应事件以及相互通信的对象。

熟悉 Java 或 C++ 中面向对象的编程 (OOP) 的程序员可能会将对象视为包含以下两类成员的模块：存储在成员变量或属性中的数据，以及可通过方法访问的行为。ECMAScript 第 4 版草案（ActionScript 3.0 所基于的标准）以相似但稍有不同的方式定义对象。在 ECMAScript 草案中，对象只是属性的集合。这些属性是一些容器，除了保存数据，还保存函数或其它对象。以这种方式附加到对象的函数称为方法。

尽管具有 Java 或 C++ 背景的程序员可能会觉得 ECMAScript 草案中的定义有些奇怪，但实际上，用 ActionScript 3.0 类定义对象类型与在 Java 或 C++ 中定义类的方式非常相似。在讨论 ActionScript 对象模型和其它高级主题时，了解这两种对象定义之间的区别很重要，但在其它大多数情况下，"属性"一词都表示类成员变量（而不是方法）。例如，《ActionScript 3.0 语言和组件参考》使用 "属性"一词表示变量或 getter-setter 属性，使用 "方法"一词表示作为类的一部分的函数。

ActionScript 中的类与 Java 或 C++ 中的类之间有一个细小的差别，那就是 ActionScript 中的类不仅仅是抽象实体。ActionScript 类由存储类的属性和方法的*类对象* 表示，这样便可以使用可能不为 Java 和 C++ 程序员所熟悉的方法，例如，在类或包的顶级包括语句或可执行代码。

ActionScript 类与 Java 或 C++ 类之间还有一个区别，那就是每个 ActionScript 类都有一个 "原型对象"。在早期的 ActionScript 版本中，原型对象链接成 "原型链"，它们共同作为整个类继承层次结构的基础。但是，在 ActionScript 3.0 中，原型对象在继承系统中仅扮演很小的角色。但是，原型对象仍非常有用，如果您希望在类的所有实例中共享某个属性及其值，可以使用原型对象来代替静态属性和方法。

过去，高级 ActionScript 程序员可以用特殊的内置语言元素来直接操作原型链。现在，由于 ActionScript 语言为基于类的编程接口提供了更成熟的实现，因此其中的许多特殊语言元素（如 `__proto__` 和 `__resolve`）不再是该语言的一部分。而且，内部继承机制的优化还排除了对继承机制的直接访问，从而大大改善了 Flash Player 的性能。

# 对象和类

在 ActionScript 3.0 中，每个对象都是由类定义的。可将类视为某一类对象的模板或蓝图。类定义中可以包括变量和常量以及方法，前者用于保存数据值，后者是封装绑定到类的行为的函数。存储在属性中的值可以是 “基元值”，也可以是其它对象。基元值是指数字、字符串或布尔值。

ActionScript 中包含许多属于核心语言的内置类。其中的某些内置类（如 Number、Boolean 和 String）表示 ActionScript 中可用的基元值。其它类（如 Array、 Math 和 XML）定义属于 ECMAScript 标准的更复杂对象。

所有的类（无论是内置类还是用户定义的类）都是从 Object 类派生的。以前在 ActionScript 方面有经验的程序员一定要注意到，Object 数据类型不再是默认的数据类型，尽管其它所有类仍从它派生。在 ActionScript 2.0 中，下面的两行代码等效，因为缺乏类型注释意味着变量为 Object 类型：

```
var someObj:Object;
var someObj;
```

但是，ActionScript 3.0 引入了无类型变量这一概念，这一类变量可通过以下两种方法来指定：

```
var someObj:*;
var someObj;
```

无类型变量与 Object 类型的变量不同。二者的主要区别在于无类型变量可以保存特殊值 `undefined`，而 Object 类型的变量则不能保存该值。

您可以使用 `class` 关键字来定义自己的类。在方法声明中，可通过以下三种方法来声明类属性 (property)：用 `const` 关键字定义常量，用 `var` 关键字定义变量，用 `get` 和 `set` 属性 (attribute) 定义 getter 和 setter 属性 (property)。可以用 `function` 关键字来声明方法。

可使用 `new` 运算符来创建类的实例。下面的示例创建 Date 类的一个名为 `myBirthday` 的实例：

```
var myBirthday:Date = new Date();
```

# 包和命名空间

包和命名空间是两个相关的概念。使用包，可以通过有利于共享代码并尽可能减少命名冲突的方式将多个类定义捆绑在一起。使用命名空间，可以控制标识符（如属性名和方法名）的可见性。无论命名空间位于包的内部还是外部，都可以应用于代码。包可用于组织类文件，命名空间可用于管理各个属性和方法的可见性。

# 包

在 ActionScript 3.0 中，包是用命名空间实现的，但包和命名空间并不同义。在声明包时，可以隐式创建一个特殊类型的命名空间并保证它在编译时是已知的。显式创建的命名空间在编译时不必是已知的。

下面的示例使用 `package` 指令来创建一个包含单个类的简单包：

```
package samples
{
  public class SampleCode
  {
    public var sampleGreeting:String;
    public function sampleFunction()
    {
      trace(sampleGreeting + " from sampleFunction()");
    }
  }
}
```

在本例中，该类的名称是 SampleCode。由于该类位于 samples 包中，因此编译器在编译时会自动将其类名称限定为完全限定名称：samples.SampleCode。编译器还限定任何属性或方法的名称，以便 `sampleGreeting` 和 `sampleFunction()` 分别变成 `samples.SampleCode.sampleGreeting` 和 `samples.SampleCode.sampleFunction()`。

许多开发人员（尤其是那些具有 Java 编程背景的人）可能会选择只将类放在包的顶级。但是，ActionScript 3.0 不但支持将类放在包的顶级，而且还支持将变量、函数甚至语句放在包的顶级。此功能的一个高级用法是，在包的顶级定义一个命名空间，以便它对于该包中的所有类均可用。但是，请注意，在包的顶级只允许使用两个访问说明符：`public` 和 `internal`。Java 允许将嵌套类声明为私有，而 ActionScript 3.0 则不同，它既不支持嵌套类也不支持私有类。

但是，在其它许多方面，ActionScript 3.0 中的包与 Java 编程语言中的包非常相似。从上一个示例可看出，完全限定的包引用点运算符 (.) 来表示，这与 Java 相同。可以用包将代码组织成直观的分层结构，以供其他程序员使用。这样，您就可以将自己所创建的包与他人共享，还可以在自己的代码中使用他人创建的包，从而推动了代码共享。

使用包还有助于确保所使用的标识符名称是唯一的，而且不与其它标识符名称冲突。事实上，有些人认为这才是包的主要优点。例如，假设两个希望相互共享代码的程序员各创建了一个名为 SampleCode 的类。如果没有包，这样就会造成名称冲突，唯一的解决方法就是重命名其中的一个类。但是，使用包，就可以将其中的一个（最好是两个）类放在具有唯一名称的包中，从而轻松地避免了名称冲突。

您还可以在包名称中嵌入点来创建嵌套包，这样就可以创建包的分层结构。 Flash Player API 提供的 flash.xml 包就是一个很好的示例。 flash.xml 包嵌套在 Flash 包中。

flash.xml 包中包含在早期的 ActionScript 版本中使用的旧 XML 分析器。该分析器现在之所以包含在 flash.xml 包中，原因之一是，旧 XML 类的名称与一个新 XML 类的名称冲突，这个新 XML 类实现 ActionScript 3.0 中的 XML for ECMAScript (E4X) 规范功能。

尽管首先将旧的 XML 类移入包中是一个不错的主意，但是旧 XML 类的大多数用户都会导入 flash.xml 包，这样，除非您总是记得使用旧 XML 类的完全限定名称 (flash.xml.XML)，否则同样会造成名称冲突。为避免这种情况，现在已将旧 XML 类命名为 XMLDocument，如下面的示例所示：

```
package flash.xml
{
  class XMLDocument {}
  class XMLNode {}
  class XMLSocket {}
}
```

大多数 Flash Player API 都划分到 flash 包中。例如，flash.display 包中包含显示列表 API，flash.events 包中包含新的事件模型。

## 创建包

ActionScript 3.0 在包、类和源文件的组织方式上具有很大的灵活性。早期的 ActionScript 版本只允许每个源文件有一个类，而且要求源文件的名称与类名称匹配。ActionScript 3.0 允许在一个源文件中包括多个类，但是，每个文件中只有一个类可供该文件外部的代码使用。换言之，每个文件中只有一个类可以在包声明中进行声明。您必须在包定义的外部声明其它任何类，以使这些类对于该源文件外部的代码不可见。在包定义内部声明的类的名称必须与源文件的名称匹配。

ActionScript 3.0 在包的声明方式上也具有更大的灵活性。在早期的 ActionScript 版本中，包只是表示可用来存放源文件的目录，您不必用 `package` 语句来声明包，而是在类声明中将包名称包括在完全限定的类名称中。在 ActionScript 3.0 中，尽管包仍表示目录，但是它现在不只包含类。在 ActionScript 3.0 中，使用 `package` 语句来声明包，这意味着您还可以在包的顶级声明变量、函数和命名空间，甚至还可以在包的顶级包括可执行语句。如果在包的顶级声明变量、函数或命名空间，则在顶级只能使用 `public` 和 `internal` 属性，并且每个文件中只能有一个包级声明使用 `public` 属性（无论该声明是类声明、变量声明、函数声明还是命名空间声明）。

包的作用是组织代码并防止名称冲突。您不应将包的概念与类继承这一不相关的概念混淆。位于同一个包中的两个类具有共同的命名空间，但是它们在其它任何方面都不必相关。同样，在语义方面，嵌套包可以与其父包无关。

## 导入包

如果您希望使用位于某个包内部的特定类，则必须导入该包或该类。这与 ActionScript 2.0 不同，在 ActionScript 2.0 中，类的导入是可选的。

以本章前面的 SampleCode 类示例为例。如果该类位于名为 samples 的包中，那么，在使用 SampleCode 类之前，您必须使用下列导入语句之一：

```
import samples.*;
```

或者

```
import samples.SampleCode;
```

通常，`import` 语句越具体越好。如果您只打算使用 samples 包中的 SampleCode 类，则应只导入 SampleCode 类，而不应导入该类所属的整个包。导入整个包可能会导致意外的名称冲突。

还必须将定义包或类的源代码放在*类路径* 内部。类路径是用户定义的本地目录路径列表，它决定了编译器将在何处搜索导入的包和类。类路径有时称为 "生成路径" 或 "源路径"。

在正确地导入类或包之后，可以使用类的完全限定名称 (samples.SampleCode)，也可以只使用类名称本身 (SampleCode)。

当同名的类、方法或属性会导致代码不明确时，完全限定的名称非常有用，但是，如果将它用于所有的标识符，则会使代码变得难以管理。例如，在实例化 SampleCode 类的实例时，使用完全限定的名称会导致代码冗长：

```
var mySample:samples.SampleCode = new samples.SampleCode();
```

包的嵌套级别越高，代码的可读性越差。如果您确信不明确的标识符不会导致问题，就可以通过使用简单的标识符来提高代码的可读性。例如，如果在实例化 SampleCode 类的新实例时仅使用类标识符，代码就会简短得多。

```
var mySample:SampleCode = new SampleCode();
```

如果您尝试使用标识符名称，而不先导入相应的包或类，编译器将找不到类定义。另一方面，即便您导入了包或类，只要尝试定义的名称与所导入的名称冲突，也会产生错误。

创建包时，该包的所有成员的默认访问说明符是 `internal`，这意味着，默认情况下，包成员仅对其所在包的其它成员可见。如果您希望某个类对包外部的代码可用，则必须将该类声明为 `public`。例如，下面的包包含 SampleCode 和 CodeFormatter 两个类：

```
// SampleCode.as 文件
package samples
{
  public class SampleCode {}
}

// CodeFormatter.as 文件
package samples
{
  class CodeFormatter {}
}
```

SampleCode 类在包的外部可见，因为它被声明为 `public` 类。但是，CodeFormatter 类仅在 samples 包的内部可见。如果您尝试访问位于 samples 包外部的 CodeFormatter 类，将会产生一个错误，如下面的示例所示：

```
import samples.SampleCode;
import samples.CodeFormatter;
var mySample:SampleCode = new SampleCode(); // 正确, public 类
var myFormatter:CodeFormatter = new CodeFormatter(); // 错误
```

如果您希望这两个类在包外部均可用，必须将它们都声明为 `public`。不能将 `public` 属性应用于包声明。

完全限定的名称可用来解决在使用包时可能发生的名称冲突。如果您导入两个包，但它们用同一个标识符来定义类，就可能会发生名称冲突。例如，请考虑下面的包，该包也有一个名为 SampleCode 的类：

```
package langref.samples
{
  public class SampleCode {}
}
```

如果按如下方式导入两个类，在引用 SampleCode 类时将会发生名称冲突：

```
import samples.SampleCode;
import langref.samples.SampleCode;
var mySample:SampleCode = new SampleCode(); // 名称冲突
```

编译器无法确定要使用哪个 SampleCode 类。要解决此冲突，必须使用每个类的完全限定名称，如下所示：

```
var sample1:samples.SampleCode = new samples.SampleCode();
var sample2:langref.samples.SampleCode = new langref.samples.SampleCode();
```

| 注意 | 具有 C++ 背景的程序员通常会将 `import` 语句与 `#include` 混淆。`#include` 指令在 C++ 中是必需的，因为 C++ 编译器一次处理一个文件，而且除非显式包括了头文件，否则将不会在其它文件中查找类定义。 ActionScript 3.0 有一个 `include` 指令，但是它的作用不是为了导入类和包。要在 ActionScript 3.0 中导入类或包，必须使用 `import` 语句，并将包含该包的源文件放在类路径中。 |
|---|---|

# 命名空间

通过命名空间可以控制所创建的属性和方法的可见性。请将 `public`、`private`、`protected` 和 `internal` 访问控制说明符视为内置的命名空间。如果这些预定义的访问控制说明符无法满足您的要求，您可以创建自己的命名空间。

如果您熟悉 XML 命名空间，那么，您对这里讨论的大部分内容不会感到陌生，但是 ActionScript 实现的语法和细节与 XML 的稍有不同。即使您以前从未使用过命名空间，也没有关系，因为命名空间概念本身很简单，但是其实现涉及一些您需要了解的特定术语。

要了解命名空间的工作方式，有必要先了解属性或方法的名称总是包含两部分：标识符和命名空间。标识符通常被视为名称。例如，以下类定义中的标识符是 sampleGreeting 和 sampleFunction()：

```
class SampleCode
{
  var sampleGreeting:String;
  function sampleFunction () {
    trace(sampleGreeting + " from sampleFunction()");
  }
}
```

只要定义不以命名空间属性开头，就会用默认 internal 命名空间限定其名称，这意味着，它们仅对同一个包中的调用方可见。如果编译器设置为严格模式，则编译器会发出一个警告，指明 internal 命名空间将应用于没有命名空间属性的任何标识符。为了确保标识符可在任何位置使用，您必须在标识符名称的前面明确加上 public 属性。在上面的示例代码中，sampleGreeting 和 sampleFunction() 都有一个命名空间值 internal。

使用命名空间时，应遵循以下三个基本步骤。第一，必须使用 namespace 关键字来定义命名空间。例如，下面的代码定义 version1 命名空间：

```
namespace version1;
```

第二，在属性或方法声明中，使用命名空间（而非访问控制说明符）来应用命名空间。下面的示例将一个名为 myFunction() 的函数放在 version1 命名空间中：

```
version1 function myFunction() {}
```

第三，在应用了该命名空间后，可以使用 use 指令引用它，也可以使用该命名空间来限定标识符的名称。下面的示例通过 use 指令来引用 myFunction() 函数：

```
use namespace version1;
myFunction();
```

您还可以使用限定名称来引用 myFunction() 函数，如下面的示例所示：

```
version1::myFunction();
```

## 定义命名空间

命名空间中包含一个名为统一资源标识符 (URI) 的值，该值有时称为*命名空间名称*。使用 URI 可确保命名空间定义的唯一性。

可通过使用以下两种方法之一来声明命名空间定义，以创建命名空间：像定义 XML 命名空间那样使用显式 URI 定义命名空间；省略 URI。下面的示例说明如何使用 URI 来定义命名空间：

```
namespace flash_proxy = “http://www.adobe.com/flash/proxy”;
```

URI 用作该命名空间的唯一标识字符串。如果您省略 URI（如下面的示例所示），则编译器将创建一个唯一的内部标识字符串来代替 URI。您对于这个内部标识字符串不具有访问权限。

```
namespace flash_proxy;
```

在定义了命名空间（具有 URI 或没有 URI）后，就不能在同一个作用域内重新定义该命名空间。如果尝试定义的命名空间以前在同一个作用域内定义过，则将生成编译器错误。

如果在某个包或类中定义了一个命名空间，则该命名空间可能对于此包或类外部的代码不可见，除非使用了相应的访问控制说明符。例如，下面的代码显示了在 flash.utils 包中定义的 `flash_proxy` 命名空间。在下面的示例中，缺乏访问控制说明符意味着 `flash_proxy` 命名空间将仅对于 flash.utils 包内部的代码可见，而对于该包外部的任何代码都不可见：

```
package flash.utils
{
  namespace flash_proxy;
}
```

下面的代码使用 `public` 属性以使 `flash_proxy` 命名空间对该包外部的代码可见：

```
package flash.utils
{
  public namespace flash_proxy;
}
```

## 应用命名空间

应用命名空间意味着在命名空间中放置定义。可以放在命名空间中的定义包括函数、变量和常量（不能将类放在自定义命名空间中）。

例如，请考虑一个使用 `public` 访问控制命名空间声明的函数。在函数的定义中使用 `public` 属性会将该函数放在 public 命名空间中，从而使该函数对于所有的代码都可用。在定义了某个命名空间之后，可以按照与使用 `public` 属性相同的方式来使用所定义的命名空间，该定义将对于可以引用您的自定义命名空间的代码可用。例如，如果您定义一个名为 `example1` 的命名空间，则可以添加一个名为 `myFunction()` 的方法并将 `example1` 用作属性，如下面的示例所示：

```
namespace example1;
class someClass
{
  example1 myFunction() {}
}
```

如果在声明 myFunction() 方法时将 example1 命名空间用作属性，则意味着该方法属于 example1 命名空间。

在应用命名空间时，应切记以下几点：

- 对于每个声明只能应用一个命名空间。
- 不能一次将同一个命名空间属性应用于多个定义。换言之，如果您希望将自己的命名空间应用于 10 个不同的函数，则必须将该命名空间作为属性分别添加到这 10 个函数的定义中。
- 如果您应用了命名空间，则不能同时指定访问控制说明符，因为命名空间和访问控制说明符是互斥的。换言之，如果应用了命名空间，就不能将函数或属性声明为 public、private、protected 或 internal。

## 引用命名空间

在使用借助于任何访问控制命名空间（如 public、private、protected 和 internal）声明的方法或属性时，无需显式引用命名空间。这是因为对于这些特殊命名空间的访问由上下文控制。例如，放在 private 命名空间中的定义会自动对于同一个类中的代码可用。但是，对于您所定义的命名空间，并不存在这样的上下文相关性。要使用已经放在某个自定义命名空间中的方法或属性，必须引用该命名空间。

可以用 use namespace 指令来引用命名空间，也可以使用名称限定符 (::) 来以命名空间限定名称。用 use namespace 指令引用命名空间会打开该命名空间，这样它便可以应用于任何未限定的标识符。例如，如果您已经定义了 example1 命名空间，则可以通过使用 use namespace example1 来访问该命名空间中的名称：

```
use namespace example1;
myFunction();
```

一次可以打开多个命名空间。在使用 use namespace 打开了某个命名空间之后，它会在打开它的整个代码块中保持打开状态。不能显式关闭命名空间。

但是，如果同时打开多个命名空间则会增加发生名称冲突的可能性。如果您不愿意打开命名空间，则可以用命名空间和名称限定符来限定方法或属性名，从而避免使用 use namespace 指令。例如，下面的代码说明如何用 example1 命名空间来限定 myFunction() 名称：

```
example1::myFunction();
```

## 使用命名空间

在 Flash Player API 中的 flash.utils.Proxy 类中，可以找到用来防止名称冲突的命名空间的实例。Proxy 类取代了 ActionScript 2.0 中的 `Object.__resolve` 属性，可用来截获对未定义的属性或方法的引用，以免发生错误。为了避免名称冲突，将 Proxy 类的所有方法都放在 `flash_proxy` 命名空间中。

为了更好地了解 `flash_proxy` 命名空间的使用方法，您需要了解如何使用 Proxy 类。Proxy 类的功能仅对于继承它的类可用。换言之，如果您要对某个对象使用 Proxy 类的方法，则该对象的类定义必须是对 Proxy 类的扩展。例如，如果您希望截获对未定义的方法的调用，则应扩展 Proxy 类，然后覆盖 Proxy 类的 `callProperty()` 方法。

前面已讲到，实现命名空间的过程通常分为三步，即定义、应用然后引用命名空间。但是，由于您从不显式调用 Proxy 类的任何方法，因此只是定义和应用 `flash_proxy` 命名空间，而从不引用它。 Flash Player API 定义 `flash_proxy` 命名空间并在 Proxy 类中应用它。在您的代码中，只需要将 `flash_proxy` 命名空间应用于扩展 Proxy 类的类。

`flash_proxy` 命名空间按照与下面类似的方法在 flash.utils 包中定义：

```
package flash.utils
{
  public namespace flash_proxy;
}
```

该命名空间将应用于 Proxy 类的方法，如下面摘自 Proxy 类的代码所示：

```
public class Proxy
{
  flash_proxy function callProperty(name:*, ... rest):*
  flash_proxy function deleteProperty(name:*):Boolean
  ...
}
```

如下面的代码所示，您必须先导入 Proxy 类和 `flash_proxy` 命名空间。随后必须声明自己的类，以便它对 Proxy 类进行扩展（如果是在严格模式下进行编译，则还必须添加 `dynamic` 属性）。在覆盖 `callProperty()` 方法时，必须使用 `flash_proxy` 命名空间。

```
package
{
  import flash.utils.Proxy;
  import flash.utils.flash_proxy;

  dynamic class MyProxy extends Proxy
  {
    flash_proxy override function callProperty(name:*, ...rest):*
    {
      trace("method call intercepted: " + name);
    }
  }
}
```

如果您创建 MyProxy 类的一个实例，并调用一个未定义的方法 （如在下面的示例中调用的 testing() 方法），Proxy 对象将截获对该方法的调用，并执行覆盖后的 callProperty() 方法内部的语句 （在本例中为一个简单的 trace() 语句）。

```
var mySample:MyProxy = new MyProxy();
mySample.testing(); // 已截获方法调用：测试
```

将 Proxy 类的方法放在 flash_proxy 命名空间内部有两个好处。第一个好处是，在扩展 Proxy 类的任何类的公共接口中，拥有单独的命名空间可提高代码的可读性。（在 Proxy 类中大约有 12 个可以覆盖的方法，所有这些方法都不能直接调用。将所有这些方法都放在公共命名空间中可能会引起混淆。）第二个好处是，当 Proxy 子类中包含名称与 Proxy 类方法的名称匹配的实例方法时，使用 flash_proxy 命名空间可避免名称冲突。例如，您可能希望将自己的某个方法命名为 callProperty()。下面的代码是可接受的，因为您所用的 callProperty() 方法位于另一个命名空间中：

```
dynamic class MyProxy extends Proxy
{
  public function callProperty() {}
  flash_proxy override function callProperty(name:*, ...rest):*
  {
    trace("method call intercepted: " + name);
  }
}
```

当您希望以一种无法由四个访问控制说明符（public、private、internal 和 protected）实现的方式提供对方法或属性的访问时，命名空间也可能会非常有用。例如，您可能有几个分散在多个包中的实用程序方法。您希望这些方法对于您的所有包均可用，但是您不希望这些方法成为公共方法。为此，您可以创建一个新的命名空间，并将它用作您自己的特殊访问控制说明符。

下面的示例使用用户定义的命名空间将两个位于不同包中的函数组合在一起。通过将它们组合到同一个命名空间中，可以通过一条 use namespace 语句使这两个函数对于某个类或某个包均可见。

本示例使用四个文件来说明此方法。所有的文件都必须位于您的类路径中。第一个文件 (myInternal.as) 用来定义 myInternal 命名空间。由于该文件位于名为 example 的包中，因此您必须将该文件放在名为 example 的文件夹中。该命名空间标记为 public，因此可以导入到其它包中。

```
// example 文件夹中的 myInternal.as
package example
{
  public namespace myInternal = "http://www.adobe.com/2006/actionscript/
  examples";
}
```

第二个文件 (Utility.as) 和第三个文件 (Helper.as) 定义的类中包含应可供其它包使用的方法。 Utility 类位于 example.alpha 包中，这意味着该文件应放在 example 文件夹下的 alpha 子文件夹中。Helper 类位于 example.beta 包中，这意味着该文件应放在 example 文件夹下的 beta 子文件夹中。这两个包（example.alpha 和 example.beta）在使用命名空间之前必须先导入它。

```
// example/alpha 文件夹中的 Utility.as
package example.alpha
{
  import example.myInternal;

  public class Utility
  {
    private static var _taskCounter:int = 0;

    public static function someTask()
    {
      _taskCounter++;
    }

    myInternal static function get taskCounter():int
    {
      return _taskCounter;
    }
  }
}

// example/beta 文件夹中的 Helper.as
package example.beta
{
  import example.myInternal;

  public class Helper
  {
    private static var _timeStamp:Date;

    public static function someTask()
    {
      _timeStamp = new Date();
    }

    myInternal static function get lastCalled():Date
    {
      return _timeStamp;
    }
  }
}
```

第四个文件 (NamespaceUseCase.as) 是主应用程序类，应是 example 文件夹的同级。在 Adobe Flash CS3 Professional 中，将此类用作 FLA 的文档类。NamespaceUseCase 类还导入 `myInternal` 命名空间，并使用它来调用位于其它包中的两个静态方法。在本示例中，使用静态方法的目的仅在于简化代码。在 `myInternal` 命名空间中既可以放置静态方法也可以放置实例方法。

```
// NamespaceUseCase.as
package
{
  import flash.display.MovieClip;
  import example.myInternal;       // 导入命名空间
  import example.alpha.Utility;    // 导入 Utility 类
  import example.beta.Helper;      // 导入 Helper 类

  public class NamespaceUseCase extends MovieClip
  {
    public function NamespaceUseCase()
    {
      use namespace myInternal;

      Utility.someTask();
      Utility.someTask();
      trace(Utility.taskCounter); // 2

      Helper.someTask();
      trace(Helper.lastCalled);   // [ 上次调用 someTask() 的时间 ]
    }
  }
}
```

# 变量

变量可用来存储程序中使用的值。要声明变量，必须将 `var` 语句和变量名结合使用。在 ActionScript 2.0 中，只有当您使用类型注释时，才需要使用 `var` 语句。在 ActionScript 3.0 中，总是需要使用 `var` 语句。例如，下面的 ActionScript 行声明一个名为 `i` 的变量：

```
var i;
```

如果在声明变量时省略了 `var` 语句，在严格模式下将出现编译器错误，在标准模式下将出现运行时错误。例如，如果以前未定义变量 `i`，则下面的代码行将产生错误：

```
i; // 如果以前未定义 i，将出错
```

要将变量与一个数据类型相关联，则必须在声明变量时进行此操作。在声明变量时不指定变量的类型是合法的，但这在严格模式下将产生编译器警告。可通过在变量名后面追加一个后跟变量类型的冒号 (:) 来指定变量类型。例如，下面的代码声明一个 int 类型的变量 `i`：

```
var i:int;
```

可以使用赋值运算符 (=) 为变量赋值。例如，下面的代码声明一个变量 i 并将值 20 赋给它:

```
var i:int;
i = 20;
```

您可能会发现在声明变量的同时为变量赋值可能更加方便，如下面的示例所示:

```
var i:int = 20;
```

通常，在声明变量的同时为变量赋值的方法不仅在赋予基元值（如整数和字符串）时很常用，而且在创建数组或实例化类的实例时也很常用。下面的示例显示了一个使用一行代码声明和赋值的数组。

```
var numArray:Array = ["zero", "one", "two"];
```

可以使用 new 运算符来创建类的实例。下面的示例创建一个名为 CustomClass 的实例，并向名为 customItem 的变量赋予对该实例的引用:

```
var customItem:CustomClass = new CustomClass();
```

如果要声明多个变量，则可以使用逗号运算符 (,) 来分隔变量，从而在一行代码中声明所有这些变量。例如，下面的代码在一行代码中声明 3 个变量:

```
var a:int, b:int, c:int;
```

也可以在同一行代码中为其中的每个变量赋值。例如，下面的代码声明 3 个变量（a、b 和 c）并为每个变量赋值:

```
var a:int = 10, b:int = 20, c:int = 30;
```

尽管您可以使用逗号运算符来将各个变量的声明组合到一条语句中，但是这样可能会降低代码的可读性。

## 了解变量的作用域

变量的 “作用域”是指可在其中通过引用词汇来访问变量的代码区域。“全局”变量是指在代码的所有区域中定义的变量，而 “局部”变量是指仅在代码的某个部分定义的变量。在 ActionScript 3.0 中，始终为变量分配声明它们的函数或类的作用域。全局变量是在任何函数或类定义的外部定义的变量。例如，下面的代码通过在任何函数的外部声明一个名为 strGlobal 的全局变量来创建该变量。从该示例可看出，全局变量在函数定义的内部和外部均可用。

```
var strGlobal:String = "Global";
function scopeTest()
{
  trace(strGlobal); // 全局
}
scopeTest();
trace(strGlobal); // 全局
```

可以通过在函数定义内部声明变量来将它声明为局部变量。可定义局部变量的最小代码区域就是函数定义。在函数内部声明的局部变量仅存在于该函数中。例如，如果在名为 localScope() 的函数中声明一个名为 str2 的变量，该变量在该函数外部将不可用。

```
function localScope()
{
  var strLocal:String = "local";
}
localScope();
trace(strLocal); // 出错，因为未在全局定义 strLocal
```

如果用于局部变量的变量名已经被声明为全局变量，那么，当局部变量在作用域内时，局部定义会隐藏（或遮蔽）全局定义。全局变量在该函数外部仍然存在。例如，下面的代码创建一个名为 str1 的全局字符串变量，然后在 scopeTest() 函数内部创建一个同名的局部变量。该函数中的 trace 语句输出该变量的局部值，而函数外部的 trace 语句则输出该变量的全局值。

```
var str1:String = "Global";
function scopeTest ()
{
  var str1:String = "Local";
  trace(str1); // 本地
}
scopeTest();
trace(str1); // 全局
```

与 C++ 和 Java 中的变量不同的是，ActionScript 变量没有块级作用域。代码块是指左大括号 ({) 与右大括号 (}) 之间的任意一组语句。在某些编程语言（如 C++ 和 Java）中，在代码块内部声明的变量在代码块外部不可用。对于作用域的这一限制称为块级作用域，ActionScript 中不存在这样的限制，如果您在某个代码块中声明一个变量，那么，该变量不仅在该代码块中可用，而且还在该代码块所属函数的其它任何部分都可用。例如，下面的函数包含在不同的块作用域中定义的变量。所有的变量均在整个函数中可用。

```
function blockTest (testArray:Array)
{
  var numElements:int = testArray.length;
  if (numElements > 0)
  {
    var elemStr:String = "Element #";
    for (var i:int = 0; i < numElements; i++)
    {
      var valueStr:String = i + ": " + testArray[i];
      trace(elemStr + valueStr);
    }
    trace(elemStr, valueStr, i);   // 仍定义了所有变量
  }
  trace(elemStr, valueStr, i); // 如果 numElements > 0，则会定义所有变量
}

blockTest(["Earth", "Moon", "Sun"]);
```

有趣的是，如果缺乏块级作用域，那么，只要在函数结束之前对变量进行声明，就可以在声明变量之前读写它。这是由于存在一种名为 “提升” 的方法，该方法表示编译器会将所有的变量声明移到函数的顶部。例如，下面的代码会进行编译，即使 num 变量的初始 trace() 函数发生在声明 num 变量之前也是如此。

```
trace(num); // NaN
var num:Number = 10;
trace(num); // 10
```

但是，编译器将不会提升任何赋值语句。这就说明了为什么 num 的初始 trace() 会生成 NaN（而非某个数字）， NaN 是 Number 数据类型变量的默认值。这意味着您甚至可以在声明变量之前为变量赋值，如下面的示例所示：

```
num = 5;
trace(num); // 5
var num:Number = 10;
trace(num); // 10
```

## 默认值

“默认值” 是在设置变量值之前变量中包含的值。首次设置变量的值实际上就是 “初始化” 变量。如果您声明了一个变量，但是没有设置它的值，则该变量便处于 “未初始化” 状态。未初始化的变量的值取决于它的数据类型。下表说明了变量的默认值，并按数据类型对这些值进行组织：

| **数据类型** | **默认值** |
|---|---|
| Boolean | false |
| int | 0 |
| Number | NaN |
| Object | null |
| String | null |
| uint | 0 |
| 未声明 （与类型注释 * 等效） | undefined |
| 其它所有类 （包括用户定义的类）。 | null |

对于 Number 类型的变量，默认值是 NaN（而非某个数字），NaN 是一个由 IEEE-754 标准定义的特殊值，它表示非数字的某个值。

如果您声明某个变量，但是未声明它的数据类型，则将应用默认数据类型 *，这实际上表示该变量是无类型变量。如果您没有用值初始化无类型变量，则该变量的默认值是 undefined。

对于 Boolean、 Number、 int 和 uint 以外的数据类型，所有未初始化变量的默认值都是 null。这适用于由 Flash Player API 定义的所有类以及您创建的所有自定义类。

对于 Boolean、Number、int 或 uint 类型的变量，`null` 不是有效值。如果您尝试将值 `null` 赋予这样的变量，则该值会转换为该数据类型的默认值。对于 Object 类型的变量，可以赋予 `null` 值。如果您尝试将值 `undefined` 赋予 Object 类型的变量，则该值会转换为 `null`。

对于 Number 类型的变量，有一个名为 `isNaN()` 的特殊的顶级函数。如果变量不是数字，该函数将返回布尔值 `true`，否则将返回 `false`。

# 数据类型

“数据类型”用来定义一组值。例如， Boolean 数据类型所定义的一组值中仅包含两个值：`true` 和 `false`。除了 Boolean 数据类型外，ActionScript 3.0 还定义了其它几个常用的数据类型，如 String、 Number 和 Array。您可以使用类或接口来自定义一组值，从而定义自己的数据类型。 ActionScript 3.0 中的所有值均是对象，而与它们是基元值还是复杂值无关。

“基元值”是一个属于下列数据类型之一的值：Boolean、int、Number、String 和 uint。基元值的处理速度通常比复杂值的处理速度快，因为 ActionScript 按照一种尽可能优化内存和提高速度的特殊方式来存储基元值。



关注技术细节的读者会发现， ActionScript 在内部将基元值作为不可改变的对象进行存储。这意味着按引用传递与按值传递同样有效。这可以减少内存的使用量并提高执行速度，因为引用通常比值本身小得多。

“复杂值”是指基元值以外的值。定义复杂值的集合的数据类型包括：Array、Date、Error、Function、 RegExp、 XML 和 XMLList。

许多编程语言都区分基元值及其包装对象。例如，Java 中有一个 int 基元值和一个包装它的 java.lang.Integer 类。Java 基元值不是对象，但它们的包装是对象，这使得基元值对于某些运算非常有用，而包装对象则更适合于其它运算。在 ActionScript 3.0 中，出于实用的目的，不对基元值及其包装对象加以区分。所有的值（甚至基元值）都是对象。 Flash Player 将这些基元类型视为特例 — 它们的行为与对象相似，但是不需要创建对象所涉及的正常开销。这意味着下面的两行代码是等效的：

```
var someInt:int = 3;
var someInt:int = new int(3);
```

上面列出的所有基元数据类型和复杂数据类型都是由 ActionScript 3.0 核心类定义的。通过 ActionScript 3.0 核心类，可以使用字面值（而非 `new` 运算符）创建对象。例如，可以使用字面值或 Array 类的构造函数来创建数组，如下所示：

```
var someArray:Array = [1, 2, 3]; // 字面值
var someArray:Array = new Array(1,2,3); // Array 构造函数
```

## 类型检查

类型检查可以在编译时或运行时执行。静态类型语言（如 C++ 和 Java）在编译时执行类型检查。动态类型语言（如 Smalltalk 和 Python）在运行时执行类型检查。ActionScript 3.0 是动态类型的语言，它在运行时执行类型检查，同时也支持在名为“严格模式”的特殊编译器模式下在编译时执行类型检查。在严格模式下，类型检查既发生在编译时也发生在运行时，但是在标准模式下，类型检查仅发生在运行时。

在构造代码时，动态类型的语言带来了极大的灵活性，但代价是在运行时可能出现类型错误。静态类型的语言在编译时报告类型错误，但代价是要求类型信息在编译时是已知的。

### 编译时类型检查

在较大的项目中通常建议使用编译时类型检查，因为随着项目变大，相对于尽早捕获类型错误，数据类型的灵活性通常会变得不那么重要。这就是为什么将 Adobe Flash CS3 Professional 和 Adobe Flex Builder 2 中的 ActionScript 编译器默认设置为在严格模式下运行的原因。

为了提供编译时类型检查，编译器需要知道代码中的变量或表达式的数据类型信息。为了显式声明变量的数据类型，请在变量名后面添加后跟数据类型的冒号运算符 (:) 作为其后缀。要将数据类型与参数相关联，应使用后跟数据类型的冒号运算符。例如，下面的代码向 `xParam` 参数中添加数据类型信息，并用显式数据类型声明变量 `myParam`:

```
function runtimeTest(xParam:String)
{
  trace(xParam);
}
var myParam:String = “hello”;
runtimeTest(myParam);
```

在严格模式下，ActionScript 编译器将类型不匹配报告为编译器错误。例如，下面的代码声明一个 Object 类型的函数参数 `xParam`，但是之后又尝试向该参数赋予 String 类型和 Number 类型的值。这在严格模式下会产生编译器错误。

```
function dynamicTest(xParam:Object)
{
  if (xParam is String)
  {
    var myStr:String = xParam; // 在严格模式下产生编译器错误
    trace("String: " + myStr);
  }
  else if (xParam is Number)
  {
    var myNum:Number = xParam; // 在严格模式下产生编译器错误
    trace("Number: " + myNum);
  }
}
```

但是，即使在严格模式下，也可以选择不在赋值语句右侧指定类型，从而退出编译时类型检查。可以通过省略类型注释或使用特殊的星号 (*) 类型注释，来将变量或表达式标记为无类型。例如，如果对上一个示例中的 xParam 参数进行修改，以便它不再有类型注释，则代码将在严格模式下进行编译：

```
function dynamicTest(xParam)
{
  if (xParam is String)
  {
    var myStr:String = xParam;
    trace("String: " + myStr);
  }
  else if (xParam is Number)
  {
    var myNum:Number = xParam;
    trace("Number: " + myNum);
  }
}
dynamicTest(100)
dynamicTest("one hundred");
```

## 运行时类型检查

在 ActionScript 3.0 中，无论是在严格模式下还是在标准模式下编译，在运行时都将进行类型检查。请考虑以下情形：将值 3 作为一个参数传递给需要数组的函数。在严格模式下，编译器将生成一个错误，因为值 3 与 Array 数据类型不兼容。如果您禁用了严格模式而在标准模式下运行，则编译器将不会指出类型不匹配，但是当 Flash Player 在运行时进行类型检查时则会产生运行时错误。

下面的示例说明一个名为 typeTest() 的函数，该函数需要一个 Array 参数，但是接收到的是值 3。在标准模式下这会产生运行时错误，因为值 3 不是参数声明的数据类型 (Array) 的成员。

```
function typeTest(xParam:Array)
{
  trace(xParam);
}
var myNum:Number = 3;
typeTest(myNum);
// 在 ActionScript 3.0 标准模式下出现运行时错误
```

还可能会出现如下情形：即使在严格模式下运行，也可能会获得运行时类型错误。如果您使用严格模式，但是通过使用无类型变量而退出了编译时类型检查，就可能会出现上述情形。当您使用无类型变量时，并不会消除类型检查，而只是将其延迟到运行时执行。例如，如果上一个示例中的 `myNum` 变量没有已声明的数据类型，那么，编译器将检测不到类型不匹配，但是 Flash Player 将生成一个运行时错误，因为它会将 `myNum` 的运行时值（赋值语句将其设置为 3）与 `xParam` 的类型（设置为 Array 数据类型）进行比较。

```
function typeTest(xParam:Array)
{
  trace(xParam);
}
var myNum = 3;
typeTest(myNum);
// ActionScript 3.0 中的运行时错误
```

与编译时类型检查相比，运行时类型检查还允许更灵活地使用继承。标准模式会将类型检查延迟到运行时执行，从而允许您引用子类的属性，即使您"上传"也是如此。当您使用基类来声明类实例的类型，但是使用子类来实例化类实例时，就会发生上传。例如，您可以创建一个名为 ClassBase 的可扩展类（具有 `final` 属性的类不能扩展）：

```
class ClassBase
{
}
```

随后，您可以创建一个名为 ClassExtender 的 ClassBase 子类，该子类具有一个名为 `someString` 的属性，如下所示：

```
class ClassExtender extends ClassBase
{
  var someString:String;
}
```

通过使用这两个类，可以创建一个使用 ClassBase 数据类型进行声明、但使用 ClassExtender 构造函数进行实例化的类实例。上传被视为安全操作，这是因为基类不包含子类中没有的任何属性或方法。

```
var myClass:ClassBase = new ClassExtender();
```

但是，子类中则包含其基类中没有的属性或方法。例如，ClassExtender 类中包含 `someString` 属性，该属性在 ClassBase 类中不存在。在 ActionScript 3.0 标准模式下，可以使用 `myClass` 实例来引用此属性，而不会生成编译时错误，如下面的示例所示：

```
var myClass:ClassBase = new ClassExtender();
myClass.someString = "hello";
// 在 ActionScript 3.0 标准模式下不出现错误
```

## is 运算符

is 运算符是 ActionScript 3.0 中的新增运算符，它可用来测试变量或表达式是否为给定数据类型的成员。在早期的 ActionScript 版本中，此功能由 instanceof 运算符提供。但在 ActionScript 3.0 中，不应使用 instanceof 运算符来测试变量或表达式是否为数据类型的成员。对于手动类型检查，应用 is 运算符来代替 instanceof 运算符，因为表达式 x instanceof y 只是在 x 的原型链中检查 y 是否存在（在 ActionScript 3.0 中，原型链不能全面地描述继承层次结构）。

is 运算符检查正确的继承层次结构，不但可以用来检查对象是否为特定类的实例，而且还可以检查对象是否是用来实现特定接口的类的实例。下面的示例创建 Sprite 类的一个名为 mySprite 的实例，并使用 is 运算符来测试 mySprite 是否为 Sprite 和 DisplayObject 类的实例，以及它是否实现 IEventDispatcher 接口：

```
var mySprite:Sprite = new Sprite();
trace(mySprite is Sprite);                // true
trace(mySprite is DisplayObject);         // true
trace(mySprite is IEventDispatcher);      // true
```

is 运算符检查继承层次结构，并正确地报告 mySprite 与 Sprite 和 DisplayObject 类兼容（Sprite 类是 DisplayObject 类的子类）。is 运算符还检查 mySprite 是否是从实现 IEventDispatcher 接口的任意类继承的。由于 Sprite 类是从实现 IEventDispatcher 接口的 EventDispatcher 类继承的，因此 is 运算符会正确地报告 mySprite 也实现该接口。

下面的示例说明与上一个示例相同的测试，但使用的是 instanceof 运算符，而不是 is 运算符。instanceof 运算符正确地识别出 mySprite 是 Sprite 或 DisplayObject 的实例，但是，当它用来测试 mySprite 是否实现 IEventDispatcher 接口时，返回的却是 false。

```
trace(mySprite instanceof Sprite);           // true
trace(mySprite instanceof DisplayObject);    // true
trace(mySprite instanceof IEventDispatcher); // false
```

## as 运算符

as 运算符是 ActionScript 3.0 中的新增运算符，也可用来检查表达式是否为给定数据类型的成员。但是，与 is 运算符不同的是，as 运算符不返回布尔值，而是返回表达式的值（代替 true）或 null（代替 false）。下面的示例说明了在简单情况下使用 as 运算符替代 is 运算符的结果，例如，检查 Sprite 实例是否为 DisplayObject、IEventDispatcher 和 Number 数据类型的成员。

```
var mySprite:Sprite = new Sprite();
trace(mySprite as Sprite); // [Sprite 对象]
trace(mySprite as DisplayObject); // [Sprite 对象]
trace(mySprite as IEventDispatcher); // [Sprite 对象]
trace(mySprite as Number);   // null
```

在使用 as 运算符时，右侧的操作数必须是数据类型。如果尝试使用表达式（而非数据类型）作为右侧的操作数，将会产生错误。

## 动态类

“动态”类定义在运行时可通过添加 / 更改属性和方法来改变的对象。非动态类（如 String 类）是“密封”类。您不能在运行时向密封类中添加属性或方法。

在声明类时，可以通过使用 `dynamic` 属性来创建动态类。例如，下面的代码创建一个名为 `Protean` 的动态类：

```
dynamic class Protean
{
  private var privateGreeting:String = "hi";
  public var publicGreeting:String = "hello";
  function Protean()
  {
    trace("Protean instance created");
  }
}
```

如果要在以后实例化 `Protean` 类的实例，则可以在类定义的外部向该类中添加属性或方法。例如，下面的代码创建 `Protean` 类的一个实例，并向该实例中添加两个名称分别为 `aString` 和 `aNumber` 的属性：

```
var myProtean:Protean = new Protean();
myProtean.aString = "testing";
myProtean.aNumber = 3;
trace(myProtean.aString, myProtean.aNumber); // 测试 3
```

添加到动态类实例中的属性是运行时实体，因此会在运行时完成所有类型检查。不能向以这种方式添加的属性中添加类型注释。

您还可以定义一个函数并将该函数附加到 `myProtean` 实例的某个属性，从而向 `myProtean` 实例中添加方法。下面的代码将 trace 语句移到一个名为 `traceProtean()` 的方法中：

```
var myProtean:Protean = new Protean();
myProtean.aString = "testing";
myProtean.aNumber = 3;
myProtean.traceProtean = function ()
{
  trace(this.aString, this.aNumber);
};
myProtean.traceProtean(); // 测试 3
```

但是，以这种方式创建的方法对于 Protean 类的任何私有属性或方法都不具有访问权限。而且，即使对 `Protean` 类的公共属性或方法的引用也必须用 `this` 关键字或类名进行限定。下面的示例说明了 `traceProtean()` 方法，该方法尝试访问 `Protean` 类的私有变量和公共变量。

```
myProtean.traceProtean = function ()
{
  trace(myProtean.privateGreeting); // undefined
  trace(myProtean.publicGreeting); // hello
};
myProtean.traceProtean();
```

# 数据类型说明

基元数据类型包括 Boolean、int、Null、Number、String、uint 和 void。ActionScript 核心类还定义下列复杂数据类型：Object、Array、Date、Error、Function、RegExp、XML 和 XMLList。

## Boolean 数据类型

Boolean 数据类型包含两个值：`true` 和 `false`。对于 Boolean 类型的变量，其它任何值都是无效的。已经声明但尚未初始化的布尔变量的默认值是 `false`。

## int 数据类型

int 数据类型在内部存储为 32 位整数，它包含一组介于 -2,147,483,648 ($-2^{31}$) 和 2,147,483,647 ($2^{31}-1$) 之间的整数（包括 -2,147,483,648 和 2,147,483,647）。早期的 ActionScript 版本仅提供 Number 数据类型，该数据类型既可用于整数又可用于浮点数。在 ActionScript 3.0 中，现在可以访问 32 位带符号整数和无符号整数的低位机器类型。如果您的变量将不会使用浮点数，那么，使用 int 数据类型来代替 Number 数据类型应会更快更高效。

对于小于 int 的最小值或大于 int 的最大值的整数值，应使用 Number 数据类型。Number 数据类型可以处理 -9,007,199,254,740,992 和 9,007,199,254,740,992（53 位整数值）之间的值。 int 数据类型的变量的默认值是 0。

## Null 数据类型

Null 数据类型仅包含一个值：`null`。这是 String 数据类型和用来定义复杂数据类型的所有类（包括 Object 类）的默认值。其它基元数据类型（如 Boolean、Number、int 和 uint）均不包含 `null` 值。如果您尝试向 Boolean、Number、int 或 uint 类型的变量赋予 `null`，则 Flash Player 会将 `null` 值转换为相应的默认值。不能将 Null 数据类型用作类型注释。

## Number 数据类型

在 ActionScript 3.0 中，Number 数据类型可以表示整数、无符号整数和浮点数。但是，为了尽可能提高性能，应将 Number 数据类型仅用于浮点数，或者用于 `int` 和 `uint` 类型可以存储的、大于 32 位的整数值。要存储浮点数，数字中应包括一个小数点。如果您省略了小数点，数字将存储为整数。

Number 数据类型使用由 IEEE 二进制浮点算术标准 (IEEE-754) 指定的 64 位双精度格式。此标准规定如何使用 64 个可用位来存储浮点数。其中的 1 位用来指定数字是正数还是负数。11 位用于指数，它以二进制的形式存储。其余的 52 位用于存储"有效位数"（又称为"尾数"），有效位数是 2 的 N 次幂， N 即前面所提到的指数。

可以将 Number 数据类型的所有位都用于有效位数，也可以将 Number 数据类型的某些位用于存储指数，后者可存储的浮点数比前者大得多。例如，如果 Number 数据类型使用全部 64 位来存储有效位数，则它可以存储的最大数字为 $2^{65}-1$。如果使用其中的 11 位来存储指数，则 Number 数据类型可以存储的最大有效数字为 $2^{1023}$。

Number 类型可以表示的最小值和最大值存储在 Number 类的名为 `Number.MAX_VALUE` 和 `Number.MIN_VALUE` 的静态属性中。

```
Number.MAX_VALUE == 1.79769313486231e+308
Number.MIN_VALUE == 4.940656458412467e-324
```

尽管这个数字范围很大，但代价是此范围的精度有所降低。Number 数据类型使用 52 位来存储有效位数，因此，那些要求用 52 位以上的位数才能精确表示的数字 （如分数 1/3）将只是近似值。如果应用程序要求小数达到绝对精度，则需要使用实现小数浮点算术（而非二进制浮点算术）的软件。

如果用 Number 数据类型来存储整数值，则仅使用 52 位有效位数。 Number 数据类型使用 52 位和一个特殊的隐藏位来表示介于 -9,007,199,254,740,992 ($-2^{53}$) 和 9,007,199,254,740,992 ($2^{53}$) 之间的整数。

Flash Player 不但将 `NaN` 值用作 `Number` 类型的变量的默认值，而且还将其用作应返回数字、却没有返回数字的任何运算的结果。例如，如果您尝试计算负数的平方根，结果将是 `NaN`。其它特殊的 Number 值包括 “正无穷大”和 “负无穷大”。



在被 0 除时，如果被除数也是 0，则结果只有一个，那就是 `NaN`。在被 0 除时，如果被除数是正数，则结果为正无穷大；如果被除数是负数，则结果为负无穷大。

## String 数据类型

String 数据类型表示一个 16 位字符的序列。字符串在内部存储为 Unicode 字符，并使用 UTF-16 格式。字符串是不可改变的值，就像在 Java 编程语言中一样。对字符串值执行运算会返回字符串的一个新实例。用 String 数据类型声明的变量的默认值是 `null`。虽然 `null` 值与空字符串 ("") 均表示没有任何字符，但二者并不相同。

## uint 数据类型

uint 数据类型在内部存储为 32 位无符号整数，它包含一组介于 0 和 4,294,967,295 ($2^{32}-1$) 之间的整数（包括 0 和 4,294,967,295）。uint 数据类型可用于要求非负整数的特殊情形。例如，必须使用 uint 数据类型来表示像素颜色值，因为 int 数据类型有一个内部符号位，该符号位并不适合处理颜色值。对于大于 uint 的最大值的整数值，应使用 Number 数据类型，该数据类型可以处理 53 位整数值。 uint 数据类型的变量的默认值是 0。

## void 数据类型

void 数据类型仅包含一个值：`undefined`。在早期的 ActionScript 版本中，`undefined` 是 Object 类实例的默认值。在 ActionScript 3.0 中， Object 实例的默认值是 `null`。如果您尝试将值 `undefined` 赋予 Object 类的实例，Flash Player 会将该值转换为 `null`。您只能为无类型变量赋予 `undefined` 这一值。无类型变量是指缺乏类型注释或者使用星号 (*) 作为类型注释的变量。只能将 `void` 用作返回类型注释。

## Object 数据类型

Object 数据类型是由 Object 类定义的。Object 类用作 ActionScript 中的所有类定义的基类。 ActionScript 3.0 中的 Object 数据类型与早期版本中的 Object 数据类型存在以下三方面的区别: 第一，Object 数据类型不再是指定给没有类型注释的变量的默认数据类型。第二， Object 数据类型不再包括 `undefined` 这一值，该值以前是 Object 实例的默认值。第三，在 ActionScript 3.0 中， Object 类实例的默认值是 `null`。

在早期的 ActionScript 版本中，会自动为没有类型注释的变量赋予 Object 数据类型。ActionScript 3.0 现在包括真正无类型变量这一概念，因此不再为没有类型注释的变量赋予 Object 数据类型。没有类型注释的变量现在被视为无类型变量。如果您希望向代码的读者清楚地表明您是故意将变量保留为无类型，可以使用新的星号 (*) 表示类型注释，这与省略类型注释等效。下面的示例显示两条等效的语句，两者都声明一个无类型变量 `x`：

```
var x
var x:*
```

只有无类型变量才能保存值 `undefined`。如果您尝试将值 `undefined` 赋给具有数据类型的变量， Flash Player 会将该值 `undefined` 转换为该数据类型的默认值。对于 Object 数据类型的实例，默认值是 `null`，这意味着，如果尝试将 `undefined` 赋给 Object 实例， Flash Player 会将值 `undefined` 转换为 `null`。

# 类型转换

在将某个值转换为其它数据类型的值时，就说发生了类型转换。类型转换可以是“隐式的”，也可以是 “显式的” 。隐式转换又称为 “强制” ，有时由 Flash Player 在运行时执行。例如，如果将值 2 赋给 Boolean 数据类型的变量，则 Flash Player 会先将值 2 转换为布尔值 `true`，然后再将其赋给该变量。显式转换又称为 “转换”，在代码指示编译器将一个数据类型的变量视为属于另一个数据类型时发生。在涉及基元值时，转换功能将一个数据类型的值实际转换为另一个数据类型的值。要将对象转换为另一类型，请用小括号括起对象名并在它前面加上新类型的名称。例如，下面的代码提取一个布尔值并将它转换为一个整数：

```
var myBoolean:Boolean = true;
var myINT:int = int(myBoolean);
trace(myINT); // 1
```

## 隐式转换

在运行时，隐式转换会发生在许多上下文中：

- 在赋值语句中
- 在将值作为函数的参数传递时
- 在从函数中返回值时
- 在使用某些运算符（如加法运算符 (+)）的表达式中

对于用户定义的类型，当要转换的值是目标类（或者派生自目标类的类）的实例时，隐式转换会成功。如果隐式转换不成功，就会出现错误。例如，下面的代码中包含成功的隐式转换和不成功的隐式转换：

```
class A {}
class B extends A {}

var objA:A = new A();
var objB:B = new B();
var arr:Array = new Array();

objA = objB; // 转换成功。
objB = arr; // 转换失败。
```

对于基元类型而言，隐式转换是通过调用内部转换算法来处理的，该算法与显式转换函数所调用的算法相同。下面各部分详细讨论了这些基元类型转换。

## 显式转换

在严格模式下进行编译时，使用显式转换会非常有用，因为您有时可能会不希望因类型不匹配而生成编译时错误。当您知道强制功能会在运行时正确转换您的值时，可能就属于这种情况。例如，在处理从表单接收的数据时，您可能希望依赖强制功能将某些字符串值转换为数值。下面的代码会生成编译时错误，即使代码在标准模式下能够正确运行也是如此：

```
var quantityField:String = "3";
var quantity:int = quantityField; // 在严格模式下出现编译时错误
```

如果您希望继续使用严格模式，但是希望将字符串转换为整数，则可以使用显式转换，如下所示：

```
var quantityField:String = "3";
var quantity:int = int(quantityField); // 显式转换成功。
```

## 转换为 int、 uint 和 Number

您可以将任何数据类型转换为以下三种数字类型之一：int、 uint 和 Number。如果 Flash Player 由于某种原因而无法转换数字，则会为 int 和 uint 数据类型赋予默认值 0，为 Number 数据类型赋予默认值 NaN。如果将布尔值转换为数字，则 true 变成值 1，false 变成值 0。

```
var myBoolean:Boolean = true;
var myUINT:uint = uint(myBoolean);
var myINT:int = int(myBoolean);
var myNum:Number = Number(myBoolean);
trace(myUINT, myINT, myNum); // 1 1 1
myBoolean = false;
myUINT = uint(myBoolean);
myINT = int(myBoolean);
myNum = Number(myBoolean);
trace(myUINT, myINT, myNum); // 0 0 0
```

仅包含数字的字符串值可以成功地转换为数字类型之一。看上去像负数的字符串或者表示十六进制值的字符串（例如，0x1A）也可以转换为数字类型。转换过程中会忽略字符串值中的前导或尾随空白字符。还可以使用 Number() 来转换看上去像浮点数的字符串。如果包含小数点，则会导致 uint() 和 int() 返回一个整数，小数点和它后面的字符被截断。例如，下面的字符串值可以转换为数字：

```
trace(uint("5"));      // 5
trace(uint("-5"));     // 4294967291。它从 MAX_VALUE 开始递减
trace(uint(" 27 "));   // 27
trace(uint("3.7"));    // 3
trace(int("3.7"));     // 3
trace(int("0x1A"));    // 26
trace(Number("3.7")); // 3.7
```

对于包含非数字字符的字符串值，在用 int() 或 uint() 转换时，将返回 0；在用 Number() 转换时，将返回 NaN。转换过程中会忽略前导和尾随空白，但是，如果字符串中包含将两个数字隔开的空白，则将返回 0 或 NaN。

```
trace(uint("5a"));     // 0
trace(uint("ten"));    // 0
trace(uint("17 63")); // 0
```

在 ActionScript 3.0 中，Number() 函数不再支持八进制数或基数为 8 的数。如果您向 ActionScript 2.0 Number() 函数提供的字符串中包含前导 0，则该数字将被视为八进制数，并将转换为等效的十进制数。对于 ActionScript 3.0 中的 Number() 函数，情况并非如此，因为 ActionScript 3.0 会忽略前导 0。例如，在使用不同的 ActionScript 版本进行编译时，下面的代码会生成不同的输出结果：

```
trace(Number("044"));
// ActionScript 3.0 44
// ActionScript 2.0 36
```

将一种数值类型的值赋给另一种数值类型的变量时，转换并不是必需的。即使在严格模式下，数值类型也会隐式转换为其它数值类型。这意味着，在某些情况下，在超出类型的范围时，可能会生成意外的值。下面的几个示例全部是在严格模式下进行编译的，但是某些示例将生成意外的值：

```
var myUInt:uint = -3; // 将 int/Number 值赋给 uint 变量
trace(myUInt); // 4294967293

var myNum:Number = sampleUINT; // 将 int/uint 值赋给 Number 变量
trace(myNum) // 4294967293

var myInt:int = uint.MAX_VALUE + 1; // 将 Number 值赋给 uint 变量
trace(myInt); // 0

myInt = int.MAX_VALUE + 1; // 将 uint/Number 值赋给 int 变量
trace(myInt); // -2147483648
```

下表概述了将其它数据类型转换为 Number、 int 或 uint 数据类型的结果。

| 数据类型或值 | 转换为 Number、 int 或 uint 时的结果 |
|---|---|
| Boolean | 如果值为 `true`，则结果为 1；否则为 0。 |
| Date | Date 对象的内部表示形式，即从 1970 年 1 月 1 日午夜 （通用时间）以来所经过的毫秒数。 |
| `null` | 0 |
| Object | 如果实例为 `null` 并转换为 Number，则结果为 `NaN`；否则为 0。 |
| String | 如果 Flash Player 可以将字符串转换为数字，则结果为数字；否则，如果转换为 Number，则结果为 `NaN`，如果转换为 int 或 uint，则结果为 0。 |
| `undefined` | 如果转换为 Number，则结果为 `NaN`；如果转换为 int 或 uint，则结果为 0。 |

## 转换为 Boolean

在从任何数值数据类型（uint、int 和 Number）转换为 Boolean 时，如果数值为 0，则结果为 `false`；否则为 `true`。对于 Number 数据类型，如果值为 `NaN`，则结果也为 `false`。下面的示例说明在转换 -1、 0 和 1 等数字时的结果：

```
var myNum:Number;
for (myNum = -1; myNum<2; myNum++)
{
  trace("Boolean(" + myNum +") is " + Boolean(myNum));
}
```

本示例的输出结果说明在这三个数字中，只有 0 返回 `false` 值：

```
Boolean(-1) is true
Boolean(0) is false
Boolean(1) is true
```

在将字符串值转换为 Boolean 数据类型时，如果字符串为 null 或空字符串 ("")，则会返回 false。否则，将返回 true。

```
var str1:String;       // 未初始化的字符串为 null。
trace(Boolean(str1));  // false

var str2:String = "";  // 空字符串
trace(Boolean(str2));  // false

var str3:String = " "; // 仅空白
trace(Boolean(str3));  // true
```

在将 Object 类的实例转换为 Boolean 数据类型时，如果该实例为 null，则将返回 false；否则将返回 true：

```
var myObj:Object;      // 未初始化的对象为 null。
trace(Boolean(myObj)); // false

myObj = new Object();  // 实例化
trace(Boolean(myObj)); // true
```

在严格模式下，系统会对布尔变量进行特殊处理，因为您不必转换即可向布尔变量赋予任何数据类型的值。即使在严格模式下，也可以将所有的数据类型隐式强制为 Boolean 数据类型。换言之，与几乎其它所有数据类型不同，转换为 Boolean 数据类型不是避免在严格模式下出错所必需的。下面的几个示例全部是在严格模式下编译的，它们在运行时按照预期的方式工作：

```
var myObj:Object = new Object();  // 实例化
var bool:Boolean = myObj;
trace(bool); // true
bool = "random string";
trace(bool); // true
bool = new Array();
trace(bool); // true
bool = NaN;
trace(bool); // false
```

下表概述了在从其它数据类型转换为 Boolean 数据类型时的结果：

| 数据类型或值 | 转换为 Boolean 数据类型时的结果 |
|---|---|
| String | 如果值为 null 或空字符串 ("")，则结果为 false；否则为 true。 |
| null | false |
| Number、int 或 uint | 如果值为 NaN 或 0，则结果为 false；否则为 true。 |
| Object | 如果实例为 null，则结果为 false；否则为 true。 |

## 转换为 String

从任何数值数据类型转换为 String 数据类型时，都会返回数字的字符串表示形式。在将布尔值转换为 String 数据类型时，如果值为 `true`，则返回字符串 `"true"`；如果值为 `false`，则返回字符串 `"false"`。

在从 Object 类的实例转换为 String 数据类型时，如果该实例为 `null`，则返回字符串 `"null"`。否则，将返回字符串 `"[object Object]"`。

在从 Array 类的实例转换为 String 时，会返回一个字符串，其中包含所有数组元素的逗号分隔列表。例如，在下面的示例中，在转换为 String 数据类型时，将返回一个包含数组中的全部三个元素的字符串：

```
var myArray:Array = ["primary", "secondary", "tertiary"];
trace(String(myArray)); // primary,secondary,tertiary
```

在从 Date 类的实例转换为 String 时，会返回该实例所包含日期的字符串表示形式。例如，下面的示例返回 Date 类实例的字符串表示形式（输出结果显示的是太平洋夏令时）：

```
var myDate:Date = new Date(2005,6,1);
trace(String(myDate)); // 星期五 7 月 1 日 00:00:00 GMT-0700 2005
```

下表概述了在将其它数据类型转换为 String 数据类型时的结果：

| 数据类型或值 | 转换为 String 数据类型时的结果 |
|---|---|
| Array | 一个包含所有数组元素的字符串。 |
| Boolean | `"true"` 或 `"false"`。 |
| Date | Date 对象的字符串表示形式。 |
| `null` | `"null"` |
| Number、 int 或 uint | 数字的字符串表示形式。 |
| Object | 如果实例为 null，则结果为 `"null"`；否则为 `"[object Object]"`。 |

# 语法

语言的语法定义了一组在编写可执行代码时必须遵循的规则。

## 区分大小写

ActionScript 3.0 是一种区分大小写的语言。只是大小写不同的标识符会被视为不同。例如，下面的代码创建两个不同的变量：

```
var num1:int;
var Num1:int;
```

## 点语法

可以通过点运算符 (.) 来访问对象的属性和方法。使用点语法，可以使用后跟点运算符和属性名或方法名的实例名来引用类的属性或方法。以下面的类定义为例：

```
class DotExample
{
  public var prop1:String;
  public function method1():void {}
}
```

借助于点语法，可以使用在如下代码中创建的实例名来访问 `prop1` 属性和 `method1()` 方法：

```
var myDotEx:DotExample = new DotExample();
myDotEx.prop1 = “hello”;
myDotEx.method1();
```

定义包时，可以使用点语法。可以使用点运算符来引用嵌套包。例如， EventDispatcher 类位于一个名为 events 的包中，该包嵌套在名为 flash 的包中。可以使用下面的表达式来引用 events 包：

```
flash.events
```

还可以使用此表达式来引用 EventDispatcher 类：

```
flash.events.EventDispatcher
```

## 斜杠语法

ActionScript 3.0 不支持斜杠语法。在早期的 ActionScript 版本中，斜杠语法用于指示影片剪辑或变量的路径。

## 字面值

“字面值”是直接出现在代码中的值。下面的示例都是字面值：

```
17
"hello"
-3
9.4
null
undefined
true
false
```

字面值还可以组合起来构成复合字面值。数组文本括在中括号字符 (`[]`) 中，各数组元素之间用逗号隔开。

数组文本可用于初始化数组。下面的几个示例显示了两个使用数组文本初始化的数组。您可以使用 `new` 语句将复合字面值作为参数传递给 Array 类构造函数，但是，您还可以在实例化下面的 ActionScript 核心类的实例时直接赋予字面值：Object、Array、String、Number、int、 uint、 XML、 XMLList 和 Boolean。

```
// 使用 new 语句。
var myStrings:Array = new Array(["alpha", "beta", "gamma"]);
var myNums:Array = new Array([1,2,3,5,8]);

// 直接赋予字面值。
var myStrings:Array = ["alpha", "beta", "gamma"];
var myNums:Array = [1,2,3,5,8];
```

字面值还可用来初始化通用对象。通用对象是 Object 类的一个实例。对象字面值括在大括号 (`{}`) 中，各对象属性之间用逗号隔开。每个属性都用冒号字符 (:) 进行声明，冒号用于分隔属性名和属性值。

可以使用 `new` 语句创建一个通用对象并将该对象的字面值作为参数传递给 Object 类构造函数，也可以在声明实例时直接将对象字面值赋给实例。下面的示例创建一个新的通用对象，并使用三个值分别设置为 1、 2 和 3 的属性 （`propA`、 `propB` 和 `propC`）初始化该对象：

```
// 使用 new 语句。
var myObject:Object = new Object({propA:1, propB:2, propC:3});

// 直接赋予字面值。
var myObject:Object = {propA:1, propB:2, propC:3};
```

有关详细信息，请参阅第 172 页的 “字符串基础知识”、第 244 页的 “正则表达式基础知识”和第 303 页的 “初始化 XML 变量”。

## 分号

可以使用分号字符 (;) 来终止语句。如果您省略分号字符，则编译器将假设每一行代码代表一条语句。由于很多程序员都习惯使用分号来表示语句结束，因此，如果您坚持使用分号来终止语句，则代码会更易于阅读。

使用分号终止语句可以在一行中放置多个语句，但是这样会使代码变得难以阅读。

## 小括号

在 ActionScript 3.0 中，可以通过三种方式来使用小括号 (`()`)。首先，可以使用小括号来更改表达式中的运算顺序。组合到小括号中的运算总是最先执行。例如，小括号可用来改变如下代码中的运算顺序：

```
trace(2 + 3 * 4);    // 14
trace( (2 + 3) * 4); // 20
```

第二，可以结合使用小括号和逗号运算符 (,) 来计算一系列表达式并返回最后一个表达式的结果，如下面的示例所示：

```
var a:int = 2;
var b:int = 3;
trace((a++, b++, a+b)); // 7
```

第三，可以使用小括号来向函数或方法传递一个或多个参数，如下面的示例所示，此示例向 trace() 函数传递一个字符串值：

```
trace("hello"); // hello
```

## 注释

ActionScript 3.0 代码支持两种类型的注释：单行注释和多行注释。这些注释机制与 C++ 和 Java 中的注释机制类似。编译器将忽略标记为注释的文本。

单行注释以两个正斜杠字符 (//) 开头并持续到该行的末尾。例如，下面的代码包含一个单行注释：

```
var someNumber:Number = 3; // 单行注释
```

多行注释以一个正斜杠和一个星号 (/*) 开头，以一个星号和一个正斜杠 (*/) 结尾。

```
/* 这是一个可以跨
多行代码的多行注释。 */
```

## 关键字和保留字

"保留字"是一些单词，因为这些单词是保留给 ActionScript 使用的，所以，不能在代码中将它们用作标识符。保留字包括 "词汇关键字"，编译器将词汇关键字从程序的命名空间中删除。如果您将词汇关键字用作标识符，则编译器会报告一个错误。下表列出了 ActionScript 3.0 词汇关键字：

| | | | |
|---|---|---|---|
| as | break | case | catch |
| class | const | continue | default |
| delete | do | else | extends |
| false | finally | for | function |
| if | implements | import | in |
| instanceof | interface | internal | is |
| native | new | null | package |
| private | protected | public | return |
| super | switch | this | throw |

| to | true | try | typeof |
| --- | --- | --- | --- |
| use | var | void | while |
| with | | | |

有一小组名为 “句法关键字” 的关键字，这些关键字可用作标识符，但是在某些上下文中具有特殊的含义。下表列出了 ActionScript 3.0 句法关键字：

| each | get | set | namespace |
| --- | --- | --- | --- |
| include | dynamic | final | native |
| override | static | | |

还有几个有时称为 “供将来使用的保留字” 的标识符。这些标识符不是为 ActionScript 3.0 保留的，但是其中的一些可能会被采用 ActionScript 3.0 的软件视为关键字。您可以在自己的代码中使用其中的许多标识符，但是 Adobe 不建议您使用它们，因为它们可能会在以后的 ActionScript 版本中作为关键字出现。

| abstract | boolean | byte | cast |
| --- | --- | --- | --- |
| char | debugger | double | enum |
| export | float | goto | intrinsic |
| long | prototype | short | synchronized |
| throws | to | transient | type |
| virtual | volatile | | |

## 常量

ActionScript 3.0 支持 `const` 语句，该语句可用来创建常量。常量是指具有无法改变的固定值的属性。只能为常量赋值一次，而且必须在最接近常量声明的位置赋值。例如，如果将常量声明为类的成员，则只能在声明过程中或者在类构造函数中为常量赋值。

下面的代码声明两个常量。第一个常量 `MINIMUM` 是在声明语句中赋值的，第二个常量 `MAXIMUM` 是在构造函数中赋值的。

```
class A
{
  public const MINIMUM:int = 0;
  public const MAXIMUM:int;

  public function A()
  {
    MAXIMUM = 10;
  }
}
```

```
var a:A = new A();
trace(a.MINIMUM); // 0
trace(a.MAXIMUM); // 10
```

如果您尝试以其它任何方法向常量赋予初始值，则会出现错误。例如，如果您尝试在类的外部设置 MAXIMUM 的初始值，将会出现运行时错误。

```
class A
{
  public const MINIMUM:int = 0;
  public const MAXIMUM:int;
}

var a:A = new A();
a["MAXIMUM"] = 10; // 运行时错误
```

Flash Player API 定义了一组广泛的常量供您使用。按照惯例，ActionScript 中的常量全部使用大写字母，各个单词之间用下划线字符 (_) 分隔。例如，MouseEvent 类定义将此命名惯例用于其常量，其中每个常量都表示一个与鼠标输入有关的事件：

```
package flash.events
{
  public class MouseEvent extends Event
  {
    public static const CLICK:String              = "click";
    public static const DOUBLE_CLICK:String       = "doubleClick";
    public static const MOUSE_DOWN:String         = "mouseDown";
    public static const MOUSE_MOVE:String         = "mouseMove";
    ...
  }
}
```

# 运算符

运算符是一种特殊的函数，它们具有一个或多个操作数并返回相应的值。“操作数”是被运算符用作输入的值，通常是字面值、变量或表达式。例如，在下面的代码中，将加法运算符 (+) 和乘法运算符 (*) 与三个字面值操作数（2、3 和 4）结合使用来返回一个值。赋值运算符 (=) 随后使用该值将所返回的值 14 赋给变量 sumNumber。

```
var sumNumber:uint = 2 + 3 * 4; // uint = 14
```

运算符可以是一元、二元或三元的。“一元”运算符有 1 个操作数。例如，递增运算符 (++) 就是一元运算符，因为它只有一个操作数。“二元”运算符有 2 个操作数。例如，除法运算符 (/) 有 2 个操作数。“三元”运算符有 3 个操作数。例如，条件运算符 (?:) 具有 3 个操作数。

有些运算符是“重载的”，这意味着它们的行为因传递给它们的操作数的类型或数量而异。例如，加法运算符 (+) 就是一个重载运算符，其行为因操作数的数据类型而异。如果两个操作数都是数字，则加法运算符会返回这些值的和。如果两个操作数都是字符串，则加法运算符会返回这两个操作数连接后的结果。下面的示例代码说明运算符的行为如何因操作数而异：

```
trace(5 + 5);      // 10
trace("5" + "5"); // 55
```

运算符的行为还可能因所提供的操作数的数量而异。减法运算符 (-) 既是一元运算符又是二元运算符。对于减法运算符，如果只提供一个操作数，则该运算符会对操作数求反并返回结果；如果提供两个操作数，则减法运算符返回这两个操作数的差。下面的示例说明首先将减法运算符用作一元运算符，然后再将其用作二元运算符。

```
trace(-3);  // -3
trace(7-2); // 5
```

## 运算符的优先级和结合律

运算符的优先级和结合律决定了运算符的处理顺序。虽然对于熟悉算术的人来说，编译器先处理乘法运算符 (*) 然后再处理加法运算符 (+) 似乎是自然而然的事情，但实际上编译器要求显式指定先处理哪些运算符。此类指令统称为“运算符优先级”。ActionScript 定义了一个默认的运算符优先级，您可以使用小括号运算符 (()) 来改变它。例如，下面的代码改变上一个示例中的默认优先级，以强制编译器先处理加法运算符，然后再处理乘法运算符：

```
var sumNumber:uint = (2 + 3) * 4; // uint == 20
```

您可能会遇到这样的情况：同一个表达式中出现两个或更多个具有相同的优先级的运算符。在这些情况下，编译器使用“结合律”的规则来确定先处理哪个运算符。除了赋值运算符之外，所有二进制运算符都是“左结合”的，也就是说，先处理左边的运算符，然后再处理右边的运算符。赋值运算符和条件运算符 (?:) 都是“右结合”的，也就是说，先处理右边的运算符，然后再处理左边的运算符。

例如，考虑小于运算符 (<) 和大于运算符 (>)，它们具有相同的优先级。如果将这两个运算符用于同一个表达式中，那么，由于这两个运算符都是左结合的，因此先处理左边的运算符。也就是说，以下两个语句将生成相同的输出结果：

```
trace(3 > 2 < 1);   // false
trace((3 > 2) < 1); // false
```

将首先处理大于运算符，这会生成值 `true`，因为操作数 3 大于操作数 2。随后，将值 `true` 与操作数 1 一起传递给小于运算符。下面的代码表示此中间状态：

```
trace((true) < 1);
```

小于运算符将值 `true` 转换为数值 1，然后将该数值与第二个操作数 1 进行比较，这将返回值 `false`（因为值 1 不小于 1）。

```
trace(1 < 1); // false
```

您可以用括号运算符来改变默认的左结合律。您可以通过用小括号括起小于运算符及其操作数来命令编译器先处理小于运算符。下面的示例使用与上一个示例相同的数，但是因为使用了小括号运算符，所以生成不同的输出结果：

```
trace(3 > (2 < 1)); // true
```

将首先处理小于运算符，这会生成值 `false`，因为操作数 2 不小于操作数 1。值 `false` 随后将与操作数 3 一起传递给大于运算符。下面的代码表示此中间状态：

```
trace(3 > (false));
```

大于运算符将值 `false` 转换为数值 0，然后将该数值与另一个操作数 3 进行比较，这将返回 `true`（因为 3 大于 0）。

```
trace(3 > 0); // true
```

下表按优先级递减的顺序列出了 ActionScript 3.0 中的运算符。该表内同一行中的运算符具有相同的优先级。在该表中，每行运算符都比位于其下方的运算符的优先级高。

| 组 | 运算符 |
|---|---|
| 主要 | `[] {x:y} () f(x) new x.y x[y] <></> @ :: ..` |
| 后缀 | `x++ x--` |
| 一元 | `++x --x + - ~ ! delete typeof void` |
| 乘法 | `* / %` |
| 加法 | `+ -` |
| 按位移位 | `<< >> >>>` |
| 关系 | `< > <= >= as in instanceof is` |
| 等于 | `== != === !==` |
| 按位“与” | `&` |
| 按位“异或” | `^` |
| 按位“或” | `\|` |
| 逻辑“与” | `&&` |
| 逻辑“或” | `\|\|` |
| 条件 | `?:` |
| 赋值 | `= *= /= %= += -= <<= >>= >>>= &= ^= \|=` |
| 逗号 | `,` |

## 主要运算符

主要运算符包括那些用来创建 Array 和 Object 字面值、对表达式进行分组、调用函数、实例化类实例以及访问属性的运算符。

下表列出了所有主要运算符，它们具有相同的优先级。属于 E4X 规范的运算符用 (E4X) 来表示。

| 运算符 | 执行的运算 |
|---|---|
| [] | 初始化数组 |
| {x:y} | 初始化对象 |
| () | 对表达式进行分组 |
| f(x) | 调用函数 |
| new | 调用构造函数 |
| x.y x[y] | 访问属性 |
| <></> | 初始化 XMLList 对象 (E4X) |
| @ | 访问属性 (E4X) |
| :: | 限定名称 (E4X) |
| .. | 访问子级 XML 元素 (E4X) |

## 后缀运算符

后缀运算符只有一个操作数，它递增或递减该操作数的值。虽然这些运算符是一元运算符，但是它们有别于其它一元运算符，被单独划归到了一个类别，因为它们具有更高的优先级和特殊的行为。在将后缀运算符用作较长表达式的一部分时，会在处理后缀运算符之前返回表达式的值。例如，下面的代码说明如何在递增值之前返回表达式 xNum++ 的值：

```
var xNum:Number = 0;
trace(xNum++); // 0
trace(xNum);   // 1
```

下表列出了所有的后缀运算符，它们具有相同的优先级：

| 运算符 | 执行的运算 |
|---|---|
| ++ | 递增 （后缀） |
| -- | 递减 （后缀） |

## 一元运算符

一元运算符只有一个操作数。这一组中的递增运算符 (++) 和递减运算符 (--) 是 "前缀运算符"，这意味着它们在表达式中出现在操作数的前面。前缀运算符与它们对应的后缀运算符不同，因为递增或递减操作是在返回整个表达式的值之前完成的。例如，下面的代码说明如何在递增值之后返回表达式 ++xNum 的值：

```
var xNum:Number = 0;
trace(++xNum); // 1
trace(xNum);   // 1
```

下表列出了所有的一元运算符，它们具有相同的优先级：

| 运算符 | 执行的运算 |
|---|---|
| ++ | 递增（前缀） |
| -- | 递减（前缀） |
| + | 一元 + |
| - | 一元 - （非） |
| ! | 逻辑 "非" |
| ~ | 按位 "非" |
| delete | 删除属性 |
| typeof | 返回类型信息 |
| void | 返回 undefined 值 |

## 乘法运算符

乘法运算符具有两个操作数，它执行乘、除或求模计算。

下表列出了所有的乘法运算符，它们具有相同的优先级：

| 运算符 | 执行的运算 |
|---|---|
| * | 乘法 |
| / | 除法 |
| % | 求模 |

## 加法运算符

加法运算符有两个操作数，它执行加法或减法计算。下表列出了所有加法运算符，它们具有相同的优先级：

| 运算符 | 执行的运算 |
| --- | --- |
| + | 加法 |
| - | 减法 |

## 按位移位运算符

按位移位运算符有两个操作数，它将第一个操作数的各位按第二个操作数指定的长度移位。下表列出了所有按位移位运算符，它们具有相同的优先级：

| 运算符 | 执行的运算 |
| --- | --- |
| << | 按位向左移位 |
| >> | 按位向右移位 |
| >>> | 按位无符号向右移位 |

## 关系运算符

关系运算符有两个操作数，它比较两个操作数的值，然后返回一个布尔值。下表列出了所有关系运算符，它们具有相同的优先级：

| 运算符 | 执行的运算 |
| --- | --- |
| < | 小于 |
| > | 大于 |
| <= | 小于或等于 |
| >= | 大于或等于 |
| as | 检查数据类型 |
| in | 检查对象属性 |
| instanceof | 检查原型链 |
| is | 检查数据类型 |

## 等于运算符

等于运算符有两个操作数，它比较两个操作数的值，然后返回一个布尔值。下表列出了所有等于运算符，它们具有相同的优先级：

| 运算符 | 执行的运算 |
|---|---|
| `==` | 等于 |
| `!=` | 不等于 |
| `===` | 严格等于 |
| `!==` | 严格不等于 |

## 按位逻辑运算符

按位逻辑运算符有两个操作数，它执行位级别的逻辑运算。按位逻辑运算符具有不同的优先级；下表按优先级递减的顺序列出了按位逻辑运算符：

| 运算符 | 执行的运算 |
|---|---|
| `&` | 按位 “与” |
| `^` | 按位 “异或” |
| `\|` | 按位 “或” |

## 逻辑运算符

逻辑运算符有两个操作数，它返回布尔结果。逻辑运算符具有不同的优先级；下表按优先级递减的顺序列出了逻辑运算符：

| 运算符 | 执行的运算 |
|---|---|
| `&&` | 逻辑 “与” |
| `\|\|` | 逻辑 “或” |

## 条件运算符

条件运算符是一个三元运算符，也就是说它有三个操作数。条件运算符是应用 `if..else` 条件语句的一种简便方法。

| 运算符 | 执行的运算 |
|---|---|
| `?:` | 条件 |

## 赋值运算符

赋值运算符有两个操作数，它根据一个操作数的值对另一个操作数进行赋值。下表列出了所有赋值运算符，它们具有相同的优先级：

| 运算符 | 执行的运算 |
|---|---|
| `=` | 赋值 |
| `*=` | 乘法赋值 |
| `/=` | 除法赋值 |
| `%=` | 求模赋值 |
| `+=` | 加法赋值 |
| `-=` | 减法赋值 |
| `<<=` | 按位向左移位赋值 |
| `>>=` | 按位向右移位赋值 |
| `>>>=` | 按位无符号向右移位赋值 |
| `&=` | 按位 “与” 赋值 |
| `^=` | 按位 “异或” 赋值 |
| `|=` | 按位 “或” 赋值 |

# 条件语句

ActionScript 3.0 提供了三个可用来控制程序流的基本条件语句。

## if..else

`if..else` 条件语句用于测试一个条件，如果该条件存在，则执行一个代码块，否则执行替代代码块。例如，下面的代码测试 `x` 的值是否超过 20，如果是，则生成一个 `trace()` 函数，否则生成另一个 `trace()` 函数：

```
if (x > 20)
{
  trace("x is > 20");
}
else
{
  trace("x is <= 20");
}
```

如果您不想执行替代代码块，可以仅使用 `if` 语句，而不用 `else` 语句。

## if..else if

可以使用 `if..else if` 条件语句来测试多个条件。例如，下面的代码不仅测试 `x` 的值是否超过 20，而且还测试 `x` 的值是否为负数：

```
if (x > 20)
{
  trace("x is > 20");
}
else if (x < 0)
{
  trace("x is negative");
}
```

如果 `if` 或 `else` 语句后面只有一条语句，则无需用大括号括起后面的语句。例如，下面的代码不使用大括号：

```
if (x > 0)
  trace("x is positive”);
else if (x < 0)
  trace("x is negative");
else
  trace("x is 0");
```

但是，Adobe 建议您始终使用大括号，因为以后在缺少大括号的条件语句中添加语句时，可能会出现意外的行为。例如，在下面的代码中，无论条件的计算结果是否为 `true`，`positiveNums` 的值总是按 1 递增：

```
var x:int;
var positiveNums:int = 0;

if (x > 0)
  trace("x is positive");
  positiveNums++;

trace(positiveNums); // 1
```

## switch

如果多个执行路径依赖于同一个条件表达式，则 `switch` 语句非常有用。它的功能大致相当于一系列 `if..else if` 语句，但是它更便于阅读。`switch` 语句不是对条件进行测试以获得布尔值，而是对表达式进行求值并使用计算结果来确定要执行的代码块。代码块以 `case` 语句开头，以 `break` 语句结尾。例如，下面的 `switch` 语句基于由 `Date.getDay()` 方法返回的日期值输出星期日期：

```
var someDate:Date = new Date();
var dayNum:uint = someDate.getDay();
switch(dayNum)
{
  case 0:
    trace("Sunday");
    break;
  case 1:
    trace("Monday");
    break;
  case 2:
    trace("Tuesday");
    break;
  case 3:
    trace("Wednesday");
    break;
  case 4:
    trace("Thursday");
    break;
  case 5:
    trace("Friday");
    break;
  case 6:
    trace("Saturday");
    break;
  default:
    trace("Out of range");
    break;
}
```

# 循环

循环语句允许您使用一系列值或变量来反复执行一个特定的代码块。 Adobe 建议您始终用大括号 ({}) 来括起代码块。尽管您可以在代码块中只包含一条语句时省略大括号，但是就像在介绍条件语言时所提到的那样，不建议您这样做，原因也相同：因为这会增加无意中将以后添加的语句从代码块中排除的可能性。如果您以后添加一条语句，并希望将它包括在代码块中，但是忘了加必要的大括号，则该语句将不会在循环过程中执行。

## for

`for` 循环用于循环访问某个变量以获得特定范围的值。必须在 `for` 语句中提供 3 个表达式：一个设置了初始值的变量，一个用于确定循环何时结束的条件语句，以及一个在每次循环中都更改变量值的表达式。例如，下面的代码循环 5 次。变量 `i` 的值从 0 开始到 4 结束，输出结果是从 0 到 4 的 5 个数字，每个数字各占 1 行。

```
var i:int;
for (i = 0; i < 5; i++)
{
  trace(i);
}
```

## for..in

`for..in` 循环用于循环访问对象属性或数组元素。例如，可以使用 `for..in` 循环来循环访问通用对象的属性（不按任何特定的顺序来保存对象的属性，因此属性可能以看似随机的顺序出现）：

```
var myObj:Object = {x:20, y:30};
for (var i:String in myObj)
{
  trace(i + ": " + myObj[i]);
}
// 输出:
// x: 20
// y: 30
```

还可以循环访问数组中的元素：

```
var myArray:Array = ["one", "two", "three"];
for (var i:String in myArray)
{
  trace(myArray[i]);
}
// 输出:
// one
// two
// three
```

如果对象是自定义类的一个实例，则除非该类是动态类，否则将无法循环访问该对象的属性。即便对于动态类的实例，也只能循环访问动态添加的属性。

## for each..in

`for each..in` 循环用于循环访问集合中的项目，它可以是 XML 或 XMLList 对象中的标签、对象属性保存的值或数组元素。例如，如下面所摘录的代码所示，您可以使用 `for each..in` 循环来循环访问通用对象的属性，但是与 `for..in` 循环不同的是，`for each..in` 循环中的迭代变量包含属性所保存的值，而不包含属性的名称：

```
var myObj:Object = {x:20, y:30};
for each (var num in myObj)
{
  trace(num);
}
// 输出:
// 20
// 30
```

您可以循环访问 XML 或 XMLList 对象，如下面的示例所示：

```
var myXML:XML = <users>
                   <fname>Jane</fname>
                   <fname>Susan</fname>
                   <fname>John</fname>
                </users>;

for each (var item in myXML.fname)
{
  trace(item);
}
/* 输出
Jane
Susan
John
*/
```

还可以循环访问数组中的元素，如下面的示例所示：

```
var myArray:Array = ["one", "two", "three"];
for each (var item in myArray)
{
  trace(item);
}
// 输出:
// one
// two
// three
```

如果对象是密封类的实例，则您将无法循环访问该对象的属性。即使对于动态类的实例，也无法循环访问任何固定属性（即，作为类定义的一部分定义的属性）。

## while

while 循环与 if 语句相似，只要条件为 true，就会反复执行。例如，下面的代码与 for 循环示例生成的输出结果相同：

```
var i:int = 0;
while (i < 5)
{
  trace(i);
  i++;
}
```

使用 while 循环（而非 for 循环）的一个缺点是，编写的 while 循环中更容易出现无限循环。如果省略了用来递增计数器变量的表达式，则 for 循环示例代码将无法编译，而 while 循环示例代码仍然能够编译。若没有用来递增 i 的表达式，循环将成为无限循环。

## do..while

do..while 循环是一种 while 循环，它保证至少执行一次代码块，这是因为在执行代码块后才会检查条件。下面的代码显示了 do...while 循环的一个简单示例，即使条件不满足，该示例也会生成输出结果：

```
var i:int = 5;
do
{
  trace(i);
  i++;
} while (i < 5);
// 输出：5
```

# 函数

“函数”是执行特定任务并可以在程序中重用的代码块。 ActionScript 3.0 中有两类函数：“方法”和 “函数闭包”。将函数称为方法还是函数闭包取决于定义函数的上下文。如果您将函数定义为类定义的一部分或者将它附加到对象的实例，则该函数称为方法。如果您以其它任何方式定义函数，则该函数称为函数闭包。

函数在 ActionScript 中始终扮演着极为重要的角色。例如，在 ActionScript 1.0 中，不存在 class 关键字，因此 “类”由构造函数定义。尽管 class 关键字已经添加到了之后的 ActionScript 版本中，但是，如果您想充分利用该语言所提供的功能，深入了解函数仍然十分重要。对于希望 ActionScript 函数的行为与 C++ 或 Java 等语言中的函数的行为相似的程序员来说，这可能是一个挑战。尽管基本的函数定义和调用对有经验的程序员来说应不是什么问题，但是仍需要对 ActionScript 函数的一些更高级功能进行解释。

# 函数的基本概念

本节讨论基本的函数定义和调用方法。

## 调用函数

可通过使用后跟小括号运算符 (()) 的函数标识符来调用函数。要发送给函数的任何函数参数都括在小括号中。例如，贯穿于本书始末的 trace() 函数，它是 Flash Player API 中的顶级函数：

```
trace("Use trace to help debug your script");
```

如果要调用没有参数的函数，则必须使用一对空的小括号。例如，可以使用没有参数的 Math.random() 方法来生成一个随机数：

```
var randomNum:Number = Math.random();
```

## 定义您自己的函数

在 ActionScript 3.0 中可通过两种方法来定义函数：使用函数语句和使用函数表达式。您可以根据自己的编程风格（偏于静态还是偏于动态）来选择相应的方法。如果您倾向于采用静态或严格模式的编程，则应使用函数语句来定义函数。如果您有特定的需求，需要用函数表达式来定义函数，则应这样做。函数表达式更多地用在动态编程或标准模式编程中。

## 函数语句

函数语句是在严格模式下定义函数的首选方法。函数语句以 function 关键字开头，后跟：

- 函数名
- 用小括号括起来的逗号分隔参数列表
- 用大括号括起来的函数体 — 即，在调用函数时要执行的 ActionScript 代码

例如，下面的代码创建一个定义一个参数的函数，然后将字符串 "hello" 用作参数值来调用该函数：

```
function traceParameter(aParam:String)
{
  trace(aParam);
}

traceParameter("hello"); // hello
```

## 函数表达式

声明函数的第二种方法就是结合使用赋值语句和函数表达式，函数表达式有时也称为函数字面值或匿名函数。这是一种较为繁杂的方法，在早期的 ActionScript 版本中广为使用。

带有函数表达式的赋值语句以 `var` 关键字开头，后跟：

- 函数名
- 冒号运算符 (:)
- 指示数据类型的 `Function` 类
- 赋值运算符 (=)
- `function` 关键字
- 用小括号括起来的逗号分隔参数列表
- 用大括号括起来的函数体 — 即，在调用函数时要执行的 ActionScript 代码

例如，下面的代码使用函数表达式来声明 `traceParameter` 函数：

```
var traceParameter:Function = function (aParam:String)
{
  trace(aParam);
};
traceParameter("hello"); // hello
```

请注意，就像在函数语句中一样，在上面的代码中也没有指定函数名。函数表达式和函数语句的另一个重要区别是，函数表达式是表达式，而不是语句。这意味着函数表达式不能独立存在，而函数语句则可以。函数表达式只能用作语句（通常是赋值语句）的一部分。下面的示例显示了一个赋予数组元素的函数表达式：

```
var traceArray:Array = new Array();
traceArray[0] = function (aParam:String)
{
  trace(aParam);
};
traceArray[0]("hello");
```

## 在函数语句和函数表达式之间进行选择

原则上，除非在特殊情况下要求使用表达式，否则应使用函数语句。函数语句较为简洁，而且与函数表达式相比，更有助于保持严格模式和标准模式的一致性。

函数语句比包含函数表达式的赋值语句更便于阅读。与函数表达式相比，函数语句使代码更为简洁而且不容易引起混淆，因为函数表达式既需要 `var` 关键字又需要 `function` 关键字。

函数语句更有助于保持严格模式和标准模式的一致性，因为在这两种编译器模式下，均可以借助点语法来调用使用函数语句声明的方法。但这对于用函数表达式声明的方法却不一定成立。例如，下面的代码定义了一个具有两个方法的 Example 类：methodExpression()（用函数表达式声明）和 methodStatement()（用函数语句声明）。在严格模式下，不能使用点语法来调用 methodExpression() 方法。

```
class Example
{
  var methodExpression = function() {}
  function methodStatement() {}
}

var myEx:Example = new Example();
myEx.methodExpression(); // 在严格模式下出错，但在标准模式下正常
myEx.methodStatement(); // 在严格模式和标准模式下均正常
```

一般认为，函数表达式更适合于关注运行时行为或动态行为的编程。如果您喜欢使用严格模式，但是还需要调用使用函数表达式声明的方法，则可以使用这两种方法中的任一方法。首先，可以使用中括号 ([]) 代替点运算符 (.) 来调用该方法。下面的方法调用在严格模式和标准模式下都能够成功执行：

```
myExample["methodLiteral"]();
```

第二，您可以将整个类声明为动态类。尽管这样您就可以使用点运算符来调用方法，但缺点是，该类的所有实例在严格模式下都将丢失一些功能。例如，如果您尝试访问动态类实例的未定义属性，则编译器不生成错误。

在某些情况下，函数表达式非常有用。函数表达式的一个常见用法就是用于那些使用一次后便丢弃的函数。另一个用法就是向原型属性附加函数，这个用法不太常见。有关详细信息，请参阅第 147 页的 “原型对象”。

函数语句与函数表达式之间有两个细微的区别，在选择要使用的方法时，应考虑这两个区别。第一个区别体现在内存管理和垃圾回收方面，因为函数表达式不像对象那样独立存在。换言之，当您将某个函数表达式分配给另一个对象（如数组元素或对象属性）时，就会在代码中创建对该函数表达式的唯一引用。如果该函数表达式所附加到的数组或对象脱离作用域或由于其它原因不再可用，您将无法再访问该函数表达式。如果删除该数组或对象，该函数表达式所使用的内存将符合垃圾回收条件，这意味着内存符合回收条件并且可重新用于其它用途。

下面的示例说明对于函数表达式，一旦删除该表达式所赋予的属性，该函数就不再可用。Test 类是动态的，这意味着您可以添加一个名为 functionExp 的属性来保存函数表达式。functionExp() 函数可以用点运算符来调用，但是一旦删除了 functionExp 属性，就无法再访问该函数。

```
dynamic class Test {}
var myTest:Test = new Test();

// 函数表达式
myTest.functionExp = function () { trace("Function expression") };
myTest.functionExp();   // 函数表达式
delete myTest.functionExp;
myTest.functionExp();   // 出错
```

另一方面，如果该函数最初是用函数语句定义的，那么，该函数将以对象的形式独立存在，即使在您删除它所附加到的属性之后，该函数仍将存在。delete 运算符仅适用于对象的属性，因此，即使是用于删除 stateFunc() 函数本身的调用也不工作。

```
dynamic class Test {}
var myTest:Test = new Test();

// 函数语句
function stateFunc() { trace("Function statement") }
myTest.statement = stateFunc;
myTest.statement(); // 函数语句
delete myTest.statement;
delete stateFunc;   // 不起作用
stateFunc();        // 函数语句
myTest.statement(); // 错误
```

函数语句与函数表达式之间的第二个区别是，函数语句存在于定义它们的整个作用域（包括出现在该函数语句前面的语句）内。与之相反，函数表达式只是为后续的语句定义的。例如，下面的代码能够在定义 scopeTest() 函数之前成功调用它：

```
statementTest(); // statementTest

function statementTest():void
{
  trace("statementTest");
}
```

函数表达式只有在定义之后才可用，因此，下面的代码会生成运行时错误：

```
expressionTest(); // 运行时错误

var expressionTest:Function = function ()
{
  trace("expressionTest");
}
```

## 从函数中返回值

要从函数中返回值，请使用后跟要返回的表达式或字面值的 return 语句。例如，下面的代码返回一个表示参数的表达式：

```
function doubleNum(baseNum:int):int
{
  return (baseNum * 2);
}
```

请注意，return 语句会终止该函数，因此，不会执行位于 return 语句下面的任何语句，如下所示：

```
function doubleNum(baseNum:int):int {
  return (baseNum * 2);
  trace("after return"); // 不会执行这条 trace 语句。
}
```

在严格模式下，如果您选择指定返回类型，则必须返回相应类型的值。例如，下面的代码在严格模式下会生成错误，因为它们不返回有效值：

```
function doubleNum(baseNum:int):int
{
  trace("after return");
}
```

## 嵌套函数

您可以嵌套函数，这意味着函数可以在其它函数内部声明。除非将对嵌套函数的引用传递给外部代码，否则嵌套函数将仅在其父函数内可用。例如，下面的代码在 getNameAndVersion() 函数内部声明两个嵌套函数：

```
function getNameAndVersion():String
{
  function getVersion():String
  {
    return "9";
  }
  function getProductName():String
  {
    return "Flash Player";
  }
  return (getProductName() + " " + getVersion());
}
trace(getNameAndVersion()); // Flash Player 9
```

在将嵌套函数传递给外部代码时，它们将作为函数闭包传递，这意味着嵌套函数保留在定义该函数时处于作用域内的任何定义。有关详细信息，请参阅第 113 页的 “函数闭包”。

## 函数参数

ActionScript 3.0 为函数参数提供了一些功能，这些功能对于那些刚接触 ActionScript 语言的程序员来说可能是很陌生的。尽管大多数程序员都应熟悉按值或按引用传递参数这一概念，但是很多人可能都对 arguments 对象和 ...(rest) 参数感到很陌生。

### 按值或按引用传递参数

在许多编程语言中，一定要了解按值传递参数与按引用传递参数之间的区别，二者之间的区别会影响代码的设计方式。

按值传递意味着将参数的值复制到局部变量中以便在函数内使用。按引用传递意味着将只传递对参数的引用，而不传递实际值。这种方式的传递不会创建实际参数的任何副本，而是会创建一个对变量的引用并将它作为参数传递，并且会将它赋给局部变量以便在函数内部使用。局部变量是对函数外部的变量的引用，它使您能够更改初始变量的值。

在 ActionScript 3.0 中，所有的参数均按引用传递，因为所有的值都存储为对象。但是，属于基元数据类型（包括 Boolean、Number、int、uint 和 String）的对象具有一些特殊运算符，这使它们可以像按值传递一样工作。例如，下面的代码创建一个名为 passPrimitives() 的函数，该函数定义了两个类型均为 int、名称分别为 xParam 和 yParam 的参数。这些参数与在 passPrimitives() 函数体内声明的局部变量类似。当使用 xValue 和 yValue 参数调用函数时， xParam 和 yParam 参数将用对 int 对象的引用进行初始化， int 对象由 xValue 和 yValue 表示。因为参数是基元值，所以它们像按值传递一样工作。尽管 xParam 和 yParam 最初仅包含对 xValue 和 yValue 对象的引用，但是，对函数体内的变量的任何更改都会导致在内存中生成这些值的新副本。

```
function passPrimitives(xParam:int, yParam:int):void
{
  xParam++;
  yParam++;
  trace(xParam, yParam);
}

var xValue:int = 10;
var yValue:int = 15;
trace(xValue, yValue);          // 10 15
passPrimitives(xValue, yValue); // 11 16
trace(xValue, yValue);          // 10 15
```

在 passPrimitives() 函数内部， xParam 和 yParam 的值递增，但这不会影响 xValue 和 yValue 的值，如上一条 trace 语句所示。即使参数的命名与 xValue 和 yValue 变量的命名完全相同也是如此，因为函数内部的 xValue 和 yValue 将指向内存中的新位置，这些位置不同于函数外部同名的变量所在的位置。

其它所有对象（即不属于基元数据类型的对象）始终按引用传递，这样您就可以更改初始变量的值。例如，下面的代码创建一个名为 objVar 的对象，该对象具有两个属性：x 和 y。该对象作为参数传递给 passByRef() 函数。因为该对象不是基元类型，所以它不但按引用传递，而且还保持一个引用。这意味着对函数内部的参数的更改将会影响到函数外部的对象属性。

```
function passByRef(objParam:Object):void
{
  objParam.x++;
  objParam.y++;
  trace(objParam.x, objParam.y);
}
var objVar:Object = {x:10, y:15};
trace(objVar.x, objVar.y); // 10 15
passByRef(objVar);         // 11 16
trace(objVar.x, objVar.y); // 11 16
```

objParam 参数与全局 objVar 变量引用相同的对象。正如在本示例的 trace 语句中所看到的一样，对 objParam 对象的 x 和 y 属性所做的更改将反映在 objVar 对象中。

## 默认参数值

ActionScript 3.0 中新增了为函数声明 “默认参数值”的功能。如果在调用具有默认参数值的函数时省略了具有默认值的参数，那么，将使用在函数定义中为该参数指定的值。所有具有默认值的参数都必须放在参数列表的末尾。指定为默认值的值必须是编译时常量。如果某个参数存在默认值，则会有效地使该参数成为 “可选参数”。没有默认值的参数被视为 “必需的参数”。

例如，下面的代码创建一个具有三个参数的函数，其中的两个参数具有默认值。当仅用一个参数调用该函数时，将使用这些参数的默认值。

```
function defaultValues(x:int, y:int = 3, z:int = 5):void
{
  trace(x, y, z);
}
defaultValues(1); // 1 3 5
```

## arguments 对象

在将参数传递给某个函数时，可以使用 arguments 对象来访问有关传递给该函数的参数的信息。arguments 对象的一些重要方面包括：

- arguments 对象是一个数组，其中包括传递给函数的所有参数。
- arguments.length 属性报告传递给函数的参数数量。
- arguments.callee 属性提供对函数本身的引用，该引用可用于递归调用函数表达式。



如果将任何参数命名为 arguments，或者使用 ...(rest) 参数，则 arguments 对象不可用。

在 ActionScript 3.0 中，函数调用中所包括的参数的数量可以大于在函数定义中所指定的参数数量，但是，如果参数的数量小于必需参数的数量，在严格模式下将生成编译器错误。您可以使用 arguments 对象的数组样式来访问传递给函数的任何参数，而无需考虑是否在函数定义中定义了该参数。下面的示例使用 arguments 数组及 arguments.length 属性来输出传递给 traceArgArray() 函数的所有参数：

```
function traceArgArray(x:int):void
{
  for (var i:uint = 0; i < arguments.length; i++)
  {
    trace(arguments[i]);
  }
}

traceArgArray(1, 2, 3);

// 输出:
// 1
// 2
// 3
```

arguments.callee 属性通常用在匿名函数中以创建递归。您可以使用它来提高代码的灵活性。如果递归函数的名称在开发周期内的不同阶段会发生改变，而且您使用的是 arguments.callee（而非函数名），则不必花费精力在函数体内更改递归调用。在下面的函数表达式中，使用 arguments.callee 属性来启用递归：

```
var factorial:Function = function (x:uint)
{
  if(x == 0)
  {
    return 1;
  }
  else
  {
    return (x * arguments.callee(x - 1));
  }
}

trace(factorial(5)); // 120
```

如果您在函数声明中使用 ...(rest) 参数，则不能使用 arguments 对象，而必须使用为参数声明的参数名来访问参数。

还应避免将 "arguments" 字符串作为参数名，因为它将遮蔽 arguments 对象。例如，如果重写 traceArgArray() 函数，以便添加 arguments 参数，那么，函数体内对 arguments 的引用所引用的将是该参数，而不是 arguments 对象。下面的代码不生成输出结果：

```
function traceArgArray(x:int, arguments:int):void
{
    for (var i:uint = 0; i < arguments.length; i++)
    {
        trace(arguments[i]);
    }
}

traceArgArray(1, 2, 3);

// 无输出
```

在早期的 ActionScript 版本中，arguments 对象还包含一个名为 caller 的属性，该属性是对当前函数的引用。 ActionScript 3.0 中没有 caller 属性，但是，如果您需要引用调用函数，则可以更改调用函数，以使其传递一个额外的参数来引用它本身。

## ...(rest) 参数

ActionScript 3.0 中引入了一个称为 ...(rest) 参数的新参数声明。此参数可用来指定一个数组参数以接受任意多个以逗号分隔的参数。此参数可以拥有保留字以外的任意名称。此参数声明必须是最后一个指定的参数。使用此参数会使 arguments 对象变得不可用。尽管 ...(rest) 参数提供了与 arguments 数组和 arguments.length 属性相同的功能，但是它不提供与 arguments.callee 类似的功能。使用 ...(rest) 参数之前，应确保不需要使用 arguments.callee。

下面的示例使用 ...(rest) 参数 （而非 arguments 对象）来重写 traceArgArray() 函数：

```
function traceArgArray(... args):void
{
    for (var i:uint = 0; i < args.length; i++)
    {
        trace(args[i]);
    }
}

traceArgArray(1, 2, 3);

// 输出：
// 1
// 2
// 3
```

...(rest) 参数还可与其它参数一起使用，前提是它是最后一个列出的参数。下面的示例修改 `traceArgArray()` 函数，以便它的第一个参数 `x` 是 int 类型，第二个参数使用 ...(rest) 参数。输出结果将忽略第一个值，因为第一个参数不再属于由 ...(rest) 参数创建的数组。

```
function traceArgArray(x: int, ... args)
{
  for (var i:uint = 0; i < args.length; i++)
  {
    trace(args[i]);
  }
}

traceArgArray(1, 2, 3);

// 输出:
// 2
// 3
```

## 函数作为对象

ActionScript 3.0 中的函数是对象。当您创建函数时，就是在创建对象，该对象不仅可以作为参数传递给另一个函数，而且还可以有附加的属性和方法。

作为参数传递给另一个函数的函数是按引用（而不是按值）传递的。在将某个函数作为参数传递时，只能使用标识符，而不能使用在调用方法时所用的小括号运算符。例如，下面的代码将名为 `clickListener()` 的函数作为参数传递给 `addEventListener()` 方法：

```
addEventListener(MouseEvent.CLICK, clickListener);
```

`Array.sort()` 方法也定义了一个接受函数的参数。有关用作 `Array.sort()` 函数参数的自定义排序函数的示例，请参阅第 194 页的 “对数组排序”。

尽管刚接触 ActionScript 的程序员可能对此感觉有些奇怪，但是，函数确实可以像其它任何对象那样具有属性和方法。实际上，每个函数都有一个名为 `length` 的只读属性，它用来存储为该函数定义的参数数量。该属性与 `arguments.length` 属性不同，后者报告发送给函数的参数数量。回想一下，在 ActionScript 中，发送给函数的参数数量可以超过为该函数定义的参数数量。下面的示例说明了这两个属性之间的区别，此示例仅在标准模式下编译，因为严格模式要求所传递的参数数量与所定义的参数数量完全相同：

```
function traceLength(x:uint, y:uint):void
{
  trace("arguments received: " + arguments.length);
  trace("arguments expected: " + traceLength.length);
}

traceLength(3, 5, 7, 11);
/* 输出:
收到的参数数量: 4
需要的参数数量: 2 */
```

您可以定义自己的函数属性，方法是在函数体外部定义它们。函数属性可以用作准静态属性，用来保存与该函数有关的变量的状态。例如，您可能希望跟踪对特定函数的调用次数。如果您正在编写游戏，并且希望跟踪用户使用特定命令的次数，则这样的功能会非常有用，尽管您也可以使用静态类属性来实现此目的。下面的代码在函数声明外部创建一个函数属性，在每次调用该函数时都递增此属性：

```
someFunction.counter = 0;

function someFunction():void
{
  someFunction.counter++;
}

someFunction();
someFunction();
trace(someFunction.counter); // 2
```

# 函数作用域

函数的作用域不但决定了可以在程序中的什么位置调用函数，而且还决定了函数可以访问哪些定义。适用于变量标识符的作用域规则同样也适用于函数标识符。在全局作用域中声明的函数在整个代码中都可用。例如，ActionScript 3.0 包含可在代码中的任意位置使用的全局函数，如 `isNaN()` 和 `parseInt()`。嵌套函数（即在另一个函数中声明的函数）可以用在声明它的函数中的任意位置。

## 作用域链

无论何时开始执行函数，都会创建许多对象和属性。首先，会创建一个称为“激活对象”的特殊对象，该对象用于存储在函数体内声明的参数以及任何局部变量或函数。由于激活对象属于内部机制，因此您无法直接访问它。接着，会创建一个“作用域链”，其中包含由 Flash Player 检查标识符声明的对象的有序列表。所执行的每个函数都有一个存储在内部属性中的作用域链。对于嵌套函数，作用域链始于其自己的激活对象，后跟其父函数的激活对象。作用域链以这种方式延伸，直到到达全局对象。全局对象是在 ActionScript 程序开始时创建的，其中包含所有的全局变量和函数。

## 函数闭包

“函数闭包”是一个对象，其中包含函数的快照及其“词汇环境”。函数的词汇环境包括函数作用域链中的所有变量、属性、方法和对象以及它们的值。无论何时在对象或类之外的位置执行函数，都会创建函数闭包。函数闭包保留定义它们的作用域，这样，在将函数作为参数或返回值传递给另一个作用域时，会产生有趣的结果。

例如，下面的代码创建两个函数：foo()（返回一个用来计算矩形面积的嵌套函数 rectArea()）和 bar()（调用 foo() 并将返回的函数闭包存储在名为 myProduct 的变量中）。即使 bar() 函数定义了自己的局部变量 x（值为 2），当调用函数闭包 myProduct() 时，该函数闭包仍保留在函数 foo() 中定义的变量 x（值为 40）。因此，bar() 函数将返回值 160，而不是 8。

```
function foo():Function
{
  var x:int = 40;
  function rectArea(y:int):int // 定义函数闭包
  {
    return x * y
  }
  return rectArea;
}
function bar():void
{
  var x:int = 2;
  var y:int = 4;
  var myProduct:Function = foo();
  trace(myProduct(4)); // 调用函数闭包
}
bar(); // 160
```

方法的行为与函数闭包类似，因为方法也保留有关创建它们的词汇环境的信息。当方法提取自它的实例（这会创建绑定方法）时，此特征尤为突出。函数闭包与绑定方法之间的主要区别在于，绑定方法中 this 关键字的值始终引用它最初附加到的实例，而函数闭包中 this 关键字的值可以改变。有关详细信息，请参阅第 128 页的 “绑定方法”。

第 4 章

# ActionScript 中面向对象的编程

4

本章介绍了支持面向对象的编程 (OOP) 的 ActionScript 元素。本章没有介绍一般 OOP 原则，如对象设计、抽象、封装、继承和多态。本章重点说明如何在使用 ActionScript 3.0 时应用这些原则。

从根本上讲，ActionScript 是一种脚本撰写语言，因此 ActionScript 3.0 OOP 支持是可选的。这为程序员提供了很大的灵活性，使他们可以为各种领域和不同复杂程度的项目选择最佳方法。对于小型任务，您可能发现使用 ActionScript 和过程化程序设计方法就能满足所有需要。对于大型项目，应用 OOP 原则可以使代码更易于理解、维护和扩展。

## 目录



# 面向对象的编程基础知识

## 面向对象的编程简介

面向对象的编程 (OOP) 是一种组织程序代码的方法，它将代码划分为对象，即包含信息（数据值）和功能的单个元素。通过使用面向对象的方法来组织程序，您可以将特定信息（例如，唱片标题、音轨标题或歌手名字等音乐信息）及其关联的通用功能或动作（如"在播放列表中添加音轨"或"播放此歌手的所有歌曲"）组合在一起。这些项目将合并为一个项目，即对象 （例如，"唱片" 或 "音轨" ）。能够将这些值和功能捆绑在一起会带来很多好处，其中包括只需跟踪单个变量而非多个变量、将相关功能组织在一起，以及能够以更接近实际情况的方式构建程序。

## 常见的面向对象编程任务

实际上，面向对象的编程包含两个部分。一部分是程序设计策略和技巧（通常称为“面向对象的设计”）。这是一个很广泛的主题，本章中不对其进行讨论。OOP 的另一部分是在给定编程语言中提供的实际编程结构，以便使用面向对象的方法来构建程序。本章介绍了 OOP 中的以下常见任务：

- 定义类
- 创建属性、方法以及 get 和 set 存取器 （存取器方法）
- 控制对类、属性、方法和存取器的访问
- 创建静态属性和方法
- 创建与枚举类似的结构
- 定义和使用接口
- 处理继承 （包括覆盖类元素）

## 重要概念和术语

以下参考列表包含将会在本章中遇到的重要术语：

- 属性 (Attribute)：在类定义中为类元素 （如属性或方法）分配的特性。属性通常用于定义程序的其它部分中的代码能否访问属性或方法。例如，`private` 和 `public` 都是属性。私有方法只能由类中的代码调用；而公共方法可以由程序中的任何代码调用。
- 类 (Class)：某种类型的对象的结构和行为定义（与该数据类型的对象的模板或蓝图类似）。
- 类层次结构 (Class hierarchy)：多个相关的类的结构，用于指定哪些类继承了其它类中的功能。
- 构造函数 (Constructor)：可以在类中定义的特殊方法，创建类的实例时将调用该方法。构造函数通常用于指定默认值，或以其它方式执行对象的设置操作。
- 数据类型 (Data type)：特定变量可以存储的信息类型。通常，“数据类型”表示与“类”相同的内容。
- 点运算符 (Dot operator)：句点符号 (.)，在 ActionScript（和很多其它编程语言）中，它用于指示某个名称引用对象的子元素（如属性或方法）。例如，在表达式 `myObject.myProperty` 中，点运算符指示 `myProperty` 项引用的值是名为 `myObject` 的对象的元素。
- 枚举 (Enumeration)：一组相关常数值，为方便起见而将其作为一个类的属性组合在一起。
- 继承 (Inheritance)：一种 OOP 机制，它允许一个类定义包含另一个类定义的所有功能（通常会添加到该功能中）。
- 实例 (Instance)：在程序中创建的实际对象。
- 命名空间 (Namespace)：实质上是一个自定义属性，它可以更精确地控制代码对其它代码的访问。

## 完成本章中的示例

学习本章的过程中，您可能想要自己动手测试一些示例代码清单。由于本章中的代码清单主要用于定义和处理数据类型，测试示例将涉及创建要定义的类的实例，使用该实例的属性或方法来处理该实例，然后查看该实例属性的值。为了查看这些值，您需要将值写入舞台上的文本字段实例，或使用 `trace()` 函数将值输出到 “输出” 面板。第 53 页的 “测试本章内的示例代码清单”中对这些技术进行了详细说明。

# 类

类是对象的抽象表示形式。类用来存储有关对象可保存的数据类型及对象可表现的行为的信息。如果编写的小脚本中只包含几个彼此交互的对象，使用这种抽象类的作用可能并不明显。但是，随着程序作用域不断扩大以及必须管理的对象数不断增加，您可能会发现，可以使用类更好地控制对象的创建方式以及对象之间的交互方式。

早在 ActionScript 1.0 中，ActionScript 程序员就能使用 Function 对象创建类似类的构造函数。在 ActionScript 2.0 中，通过使用 `class` 和 `extends` 等关键字，正式添加了对类的支持。ActionScript 3.0 不但继续支持 ActionScript 2.0 中引入的关键字，而且还添加了一些新功能，如通过 `protected` 和 `internal` 属性增强了访问控制，通过 `final` 和 `override` 关键字增强了对继承的控制。

如果您曾经使用类似 Java、 C++ 或 C# 这样的编程语言创建过类，就会发现 ActionScript 中的实现方法与之类似。 ActionScript 共享了许多相同的关键字和属性名，如 `class`、`extends` 和 `public`，以下各部分将讨论所有这些内容。



本章中，术语 “属性” 表示对象或类的任何成员，包括变量、常量和方法。此外，虽然术语 “类” 和 “静态” 经常互换使用，但在本章中这两个术语的概念是不同的。例如，本章中短语 “类属性” 指的是类的所有成员，而不仅是静态成员。

## 类定义

ActionScript 3.0 类定义使用的语法与 ActionScript 2.0 类定义使用的语法相似。正确的类定义语法中要求 `class` 关键字后跟类名。类体要放在大括号 (`{}`) 内，且放在类名后面。例如，以下代码创建了名为 Shape 的类，其中包含名为 `visible` 的变量：

```
public class Shape
{
  var visible:Boolean = true;
}
```

对于包中的类定义，有一项重要的语法更改。在 ActionScript 2.0 中，如果类在包中，则在类声明中必须包含包名称。在 ActionScript 3.0 中，引入了 package 语句，包名称必须包含在包声明中，而不是包含在类声明中。例如，以下类声明说明如何在 ActionScript 2.0 和 ActionScript 3.0 中定义 BitmapData 类（该类是 flash.display 包的一部分）：

```
// ActionScript 2.0
class flash.display.BitmapData {}

// ActionScript 3.0
package flash.display
{
  public class BitmapData {}
}
```

## 类属性

在 ActionScript 3.0 中，可使用以下四个属性之一来修改类定义：

| 属性 | 定义 |
|---|---|
| dynamic | 允许在运行时向实例添加属性。 |
| final | 不得由其它类扩展。 |
| internal（默认） | 对当前包内的引用可见。 |
| 公共 | 对所有位置的引用可见。 |

使用 internal 以外的每个属性时，必须显式包含该属性才能获得相关的行为。例如，如果定义类时未包含 dynamic 属性 (attribute)，则不能在运行时向类实例中添加属性 (property)。通过在类定义的开始处放置属性，可显式地分配属性，如下面的代码所示：

```
dynamic class Shape {}
```

请注意，列表中未包含名为 abstract 的属性。这是因为 ActionScript 3.0 不支持抽象类。同时还请注意，列表中未包含名为 private 和 protected 的属性。这些属性只在类定义中有意义，但不可以应用于类本身。如果不希望某个类在包以外公开可见，请将该类放在包中，并用 internal 属性标记该类。或者，可以省略 internal 和 public 这两个属性，编译器会自动为您添加 internal 属性。如果不希望某个类在定义该类的源文件以外可见，请将类放在包定义右大括号下面的源文件底部。

## 类体

类体放在大括号内，用于定义类的变量、常量和方法。下面的示例显示 Adobe Flash Player API 中 Accessibility 类的声明：

```
public final class Accessibility
{
  public static function get active():Boolean;
  public static function updateProperties():void;
}
```

还可以在类体中定义命名空间。下面的示例说明如何在类体中定义命名空间，以及如何在该类中将命名空间用作方法的属性：

```
public class SampleClass
{
  public namespace sampleNamespace;
  sampleNamespace function doSomething():void;
}
```

ActionScript 3.0 不但允许在类体中包括定义，而且还允许包括语句。如果语句在类体中但在方法定义之外，这些语句只在第一次遇到类定义并且创建了相关的类对象时执行一次。下面的示例包括一个对 `hello()` 外部函数的调用和一个 `trace` 语句，该语句在定义类时输出确认消息：

```
function hello():String
{
  trace("hola");
}
class SampleClass
{
  hello();
  trace("class created");
}
// 创建类时输出
hola
class created
```

与以前版本的 ActionScript 相比， ActionScript 3.0 中允许在同一类体中定义同名的静态属性和实例属性。例如，下面的代码声明一个名为 `message` 的静态变量和一个同名的实例变量：

```
class StaticTest
{
  static var message:String = "static variable";
  var message:String = "instance variable";
}
// 在脚本中
var myST:StaticTest = new StaticTest();
trace(StaticTest.message); // 输出：静态变量
trace(myST.message);       // 输出：实例变量
```

## 类属性 (property) 的属性 (attribute)

讨论 ActionScript 对象模型时，术语 "属性" 指可以成为类成员的任何成员，包括变量、常量和方法。这与《ActionScript 3.0 语言和组件参考》中该术语的使用方式有所不同，后者中该术语的使用范围更窄，只包括作为变量的类成员或用 getter 或 setter 方法定义的类成员。在 ActionScript 3.0 中，提供了可以与类的任何属性 (property) 一起使用的一组属性 (attribute)。下表列出了这组属性。

| 属性 | 定义 |
|---|---|
| `internal` （默认） | 对同一包中的引用可见。 |
| `private` | 对同一类中的引用可见。 |
| `protected` | 对同一类及派生类中的引用可见。 |
| `public` | 对所有位置的引用可见。 |
| `static` | 指定某一属性属于该类，而不属于该类的实例。 |
| *`UserDefinedNamespace`* | 用户定义的自定义命名空间名。 |

## 访问控制命名空间属性

ActionScript 3.0 提供了四个特殊的属性 (attribute) 来控制对在类中定义的属性 (property) 的访问：`public`、`private`、`protected` 和 `internal`。

使用 `public` 属性 (attribute) 可使某一属性 (property) 在脚本的任何位置可见。例如，要使某个方法可用于包外部的代码，必须使用 `public` 属性声明该方法。这适用于任何属性，不管属性是使用 `var`、`const` 还是 `function` 关键字声明的。

使用 `private` 属性 (attribute) 可使某一属性 (property) 只对属性 (property) 的定义类中的调用方可见。这一行为不同于 ActionScript 2.0 中 `private` 属性 (attribute) 的行为，后者允许子类访问超类中的私有属性 (property)。另一处明显的行为变化是必须执行运行时访问。在 ActionScript 2.0 中，`private` 关键字只在编译时禁止访问，运行时很容易避开它。在 ActionScript 3.0 中，这种情况不复存在。标记为 `private` 的属性在编译时和运行时都不可用。

例如，下面的代码创建了名为 PrivateExample 的简单类，其中包含一个私有变量，然后尝试从该类的外部访问该私有变量。在 ActionScript 2.0 中，编译时访问被禁止，但是使用属性访问运算符 (`[]`) 可以很容易地避开禁止，属性访问运算符在运行时 （而不是编译时）执行属性查找。

```
class PrivateExample
{
  private var privVar:String = "private variable";
}

var myExample:PrivateExample = new PrivateExample();
```

```
trace(myExample.privVar);    // 在严格模式下发生编译时错误
trace(myExample["privVar"]); // ActionScript 2.0 允许访问，但在 ActionScript
   3.0 中，这是一个运行时错误。
```

在 ActionScript 3.0 中使用严格模式时，尝试使用点运算符 (myExample.privVar) 访问私有属性会导致编译时错误。否则，会在运行时报告错误，就像使用属性访问运算符 (myExample["privVar"]) 时一样。

下表汇总了试图访问属于密封（非动态）类的 private 属性的结果：

| | 严格模式 | 标准模式 |
|---|---|---|
| 点运算符 (.) | 编译时错误 | 运行时错误 |
| 中括号运算符 ([]) | 运行时错误 | 运行时错误 |

在使用 dynamic 属性声明的类中尝试访问私有变量时，不会导致运行时错误。只是变量不可见，所以 Flash Player 返回值 undefined。但是，如果在严格模式下使用点运算符，则会发生编译时错误。下面的示例与上一个示例相同，只是 PrivateExample 类被声明为动态类：

```
dynamic class PrivateExample
{
   private var privVar:String = "private variable";
}

var myExample:PrivateExample = new PrivateExample();
trace(myExample.privVar);    // 在严格模式下发生编译时错误
trace(myExample["privVar"]); // 输出：undefined
```

当类外部的代码尝试访问 private 属性时，动态类通常会返回值 undefined，而不是生成错误。下表说明了只有在严格模式下使用点运算符访问 private 属性时才会生成错误：

| | 严格模式 | 标准模式 |
|---|---|---|
| 点运算符 (.) | 编译时错误 | undefined |
| 中括号运算符 ([]) | undefined | undefined |

protected 属性 (attribute) 是 ActionScript 3.0 中的新增属性 (attribute)，可使属性 (property) 对所属类或子类中的调用方可见。换句话说， protected 属性在所属类中可用，或者对继承层次结构中该类下面的类可用。无论子类在同一包中还是在不同包中，这一点都适用。

对于熟悉 ActionScript 2.0 的用户而言，此功能类似于 ActionScript 2.0 中的 private 属性。ActionScript 3.0 中的 protected 属性还类似于 Java 中的 protected 属性，不同之处在于，在 Java 版中该属性还允许访问同一包中的调用方。如果存在子类需要的变量或方法，但要对继承链外部的代码隐藏该变量或方法，此时 protected 属性会很有用。

`internal` 属性 (attribute) 是 ActionScript 3.0 的新增属性 (attribute)，可使属性 (property) 对所在包中的调用方可见。该属性是包中代码的默认属性 (attribute)，它适用于没有以下任意属性 (attribute) 的任何属性 (property)：

- `public`
- `private`
- `protected`
- 用户定义的命名空间

`internal` 属性与 Java 中的默认访问控制相似，不过，在 Java 中该访问级别没有明确的名称，只能通过省略所有其它访问修饰符来实现。 ActionScript 3.0 中提供的 `internal` 属性 (attribute) 旨在为您提供一个明确表达自己意图的选项，以使属性 (property) 仅对所在包中的调用方可见。

## static 属性

`static` 属性 (attribute) 可以与用 `var`、`const` 或 `function` 关键字声明的那些属性 (property) 一起使用，使用该属性 (attribute) 可将属性 (property) 附加到类而不是类的实例。类外部的代码必须使用类名（而不是使用实例名）调用静态属性 (property)。

静态属性不由子类继承的，但是这些属性是子类作用域链中的一部分。这意味着在子类体中，不必引用在其中定义静态变量或方法的类，就可以使用静态变量或方法。有关详细信息，请参阅第 140 页的 “不继承静态属性”。

## 用户定义的命名空间属性

作为预定义访问控制属性的替代方法，您可以创建自定义命名空间以用作属性。每个定义只能使用一个命名空间属性，而且不能将命名空间属性与任何访问控制属性（`public`、`private`、`protected` 和 `internal`）组合使用。有关使用命名空间的详细信息，请参阅第 61 页的 “命名空间”。

# 变量

可以使用 `var` 或 `const` 关键字声明变量。在脚本的整个执行过程中，使用 `var` 关键字声明的变量可多次更改其变量值。使用 `const` 关键字声明的变量称为 “常量”，只能赋值一次。尝试给已初始化的常量分配新值，将生成错误。有关详细信息，请参阅第 89 页的 “常量”。

## 静态变量

静态变量是使用 static 关键字和 var 或 const 语句共同声明的。静态变量附加到类而不是类的实例，对于存储和共享应用于对象的整个类的信息非常有用。例如，当要保存类实例化的总次数或者要存储允许的最大类实例数，使用静态变量比较合适。

下面的示例创建一个 totalCount 变量（用于跟踪类实例化数）和一个 MAX_NUM 常量（用于存储最大实例化数）。totalCount 和 MAX_NUM 这两个变量是静态变量，因为它们包含的值应用于整个类，而不是某个特定实例。

```
class StaticVars
{
    public static var totalCount:int = 0;
    public static const MAX_NUM:uint = 16;
}
```

StaticVars 类及其任何子类外部的代码只能通过该类本身来引用 totalCount 和 MAX_NUM 属性。例如，以下代码有效：

```
trace(StaticVars.totalCount); // 输出: 0
trace(StaticVars.MAX_NUM); // 输出: 16
```

不能通过类实例访问静态变量，因此以下代码会返回错误：

```
var myStaticVars:StaticVars = new StaticVars();
trace(myStaticVars.totalCount); // 错误
trace(myStaticVars.MAX_NUM); // 错误
```

必须在声明常量的同时初始化使用 static 和 const 关键字声明的变量，就像 StaticVars 类初始化 MAX_NUM 那样。您不能为构造函数或实例方法中的 MAX_NUM 赋值。以下代码会生成错误，因为它不是初始化静态常量的有效方法：

```
// !! 采用这种方法初始化静态常量将发生错误
class StaticVars2
{
    public static const UNIQUESORT:uint;
    function initializeStatic():void
    {
        UNIQUESORT = 16;
    }
}
```

## 实例变量

实例变量包括使用 var 和 const 关键字但未使用 static 关键字声明的属性。实例变量附加到类实例而不是整个类，对于存储特定于实例的值很有用。例如， Array 类有一个名为 length 的实例属性，用来存储 Array 类的特定实例保存的数组元素的个数。

不能覆盖子类中声明为 var 或 const 的实例变量。但是，通过覆盖 getter 和 setter 方法，可以实现类似于覆盖变量的功能。有关详细信息，请参阅第 127 页的 “get 和 set 存取器方法”。

# 方法

方法是类定义中的函数。创建类的一个实例后，该实例就会捆绑一个方法。与在类外部声明的函数不同，不能将方法与附加方法的实例分开使用。

方法是使用 function 关键字定义的。您可以使用函数语句，如下所示：

```
public function sampleFunction():String {}
```

或者，也可以使用分配了函数表达式的变量，如下所示：

```
public var sampleFunction:Function = function () {}
```

多数情况下，您需要使用函数语句而不是函数表达式，原因如下：

- 函数语句更为简洁易读。
- 函数语句允许使用 override 和 final 关键字。有关详细信息，请参阅第 138 页的 “覆盖方法”。
- 函数语句在标识符 （即函数名）与方法体代码之间创建了更强的绑定。由于可以使用赋值语句更改变量值，可随时断开变量与其函数表达式之间的连接。虽然可通过使用 const （不是 var）声明变量来解决这个问题，但这种方法并不是最好的做法，因为这会使代码难以阅读，还会禁止使用 override 和 final 关键字。

必须使用函数表达式的一种情况是：选择将函数附加到原型对象时。有关详细信息，请参阅第 147 页的 “原型对象”。

## 构造函数方法

构造函数方法有时简单称为 “构造函数”，是与在其中定义函数的类共享同一名称的函数。只要使用 new 关键字创建了类实例，就会执行构造函数方法中包括的所有代码。例如，以下代码定义名为 Example 的简单类，该类包含名为 status 的属性。 status 变量的初始值是在构造函数中设置的。

```
class Example
{
  public var status:String;
  public function Example()
  {
    status = "initialized";
  }
}

var myExample:Example = new Example();
trace(myExample.status); // 输出: 已初始化
```

构造函数方法只能是公共方法，但可以选择性地使用 public 属性。不能对构造函数使用任何其它访问控制说明符（包括使用 private、protected 或 internal）。也不能对函数构造方法使用用户定义的命名空间。

构造函数可以使用 super() 语句显式地调用其直接超类的构造函数。如果未显式调用超类构造函数，编译器会在构造函数体中的第一个语句前自动插入一个调用。还可以使用 super 前缀作为对超类的引用来调用超类的方法。如果决定在同一构造函数中使用 super() 和 super，务必先调用 super()。否则，super 引用的行为将会与预期不符。另外，super() 构造函数也应在 throw 或 return 语句之前调用。

下面的示例说明如果在调用 super() 构造函数之前尝试使用 super 引用，将会发生什么情况。新类 ExampleEx 扩展了 Example 类。ExampleEx 构造函数尝试访问在其超类中定义的状态变量，但访问是在调用 super() 之前进行的。ExampleEx 构造函数中的 trace() 语句生成了 null 值，原因是 status 变量在 super() 构造函数执行之前不可用。

```
class ExampleEx extends Example
{
    public function ExampleEx()
    {
        trace(super.status);
        super();
    }
}

var mySample:ExampleEx = new ExampleEx(); // 输出: null
```

虽然在构造函数中使用 return 语句是合法的，但是不允许返回值。换句话说，return 语句不得有相关的表达式或值。因此，不允许构造函数方法返回值，这意味着不可以指定任何返回值。

如果没有在类中定义构造函数方法，编译器将会为您自动创建一个空构造函数。如果某个类扩展了另一个类，编译器将会在所生成的构造函数中包括 super() 调用。

## 静态方法

静态方法也叫做 "类方法"，它们是使用 static 关键字声明的方法。静态方法附加到类而不是类的实例，因此在封装对单个实例的状态以外的内容有影响的功能时，静态方法很有用。由于静态方法附加到整个类，所以只能通过类访问静态方法，而不能通过类实例访问。

静态方法为封装所提供的功能不仅仅在影响类实例状态的方面。换句话说，如果方法提供的功能对类实例的值没有直接的影响，该方法应是静态方法。例如，Date 类具有名为 parse() 的静态方法，它接收字符串并将其转换为数字。该方法就是静态方法，因为它并不影响类的单个实例。而 parse() 方法使用表示日期值的字符串，分析该字符串，然后使用与 Date 对象的内部表示形式兼容的格式返回一个数字。此方法不是实例方法，因为将该方法应用到 Date 类的实例并没有任何意义。

请将静态 parse() 方法与 Date 类的一个实例方法（如 getMonth()）相比较。getMonth() 方法是一个实例方法，因为它通过检索 Date 实例的特定组件（即 month），对实例值直接执行操作。

由于静态方法不绑定到单个实例，因此不能在静态方法体中使用关键字 this 或 super。this 和 super 这两个引用只在实例方法上下文中有意义。

与其它基于类的编程语言不同， ActionScript 3.0 中的静态方法不是继承的。有关详细信息，请参阅第 140 页的 “不继承静态属性”。

## 实例方法

实例方法指的是不使用 static 关键字声明的方法。实例方法附加到类实例而不是整个类，在实现对类的各个实例有影响的功能时，实例方法很有用。例如，Array 类包含名为 sort() 的实例方法，该实例方法直接对 Array 实例执行操作。

在实例方法体中，静态变量和实例变量都在作用域中，这表示使用一个简单的标识符可以引用同一类中定义的变量。例如，以下类 CustomArray 扩展了 Array 类。CustomArray 类定义一个名为 arrayCountTotal 的静态变量 （用于跟踪类实例总数）、一个名为 arrayNumber 实例变量（用于跟踪创建实例的顺序）和一个名为 getPosition() 的实例方法（用于返回这两个变量的值）。

```
public class CustomArray extends Array
{
  public static var arrayCountTotal:int = 0;
  public var arrayNumber:int;

  public function CustomArray()
  {
    arrayNumber = ++arrayCountTotal;
  }

  public function getArrayPosition():String
  {
     return ("Array " + arrayNumber + " of " + arrayCountTotal);
  }
}
```

虽然类外部的代码必须使用 CustomArray.arrayCountTotal 通过类对象来引用 arrayCountTotal 静态变量，但是位于 getPosition() 方法体中的代码可以直接引用静态 arrayCountTotal 变量。即使对于超类中的静态变量，这一点也适用。虽然在 ActionScript 3.0 中不继承静态属性，但是超类的静态属性在作用域中。例如， Array 类有几个静态变量，其中一个是名为 DESCENDING 的常量。位于 Array 子类中的代码可以使用一个简单的标识符来引用静态常量 DESCENDING。

```
public class CustomArray extends Array
{
  public function testStatic():void
  {
    trace(DESCENDING); // 输出: 2
  }
}
```

实例方法体中的 this 引用的值是对方法所附加实例的引用。下面的代码说明 this 引用指向包含方法的实例：

```
class ThisTest
{
  function thisValue():ThisTest
  {
    return this;
  }
}

var myTest:ThisTest = new ThisTest();
trace(myTest.thisValue() == myTest); // 输出: true
```

使用关键字 override 和 final 可以控制实例方法的继承。可以使用 override 属性重新定义继承的方法，以及使用 final 属性禁止子类覆盖方法。有关详细信息，请参阅第 138 页的 "覆盖方法"。

## get 和 set 存取器方法

get 和 set 存取器函数还分别称为 *getter* 和 *setter*，可以使用这些函数为创建的类提供易于使用的编程接口，并遵循信息隐藏和封装的编程原则。使用 get 和 set 函数可保持类的私有类属性，但允许类用户访问这些属性，就像他们在访问类变量而不是调用类方法。

这种方法的好处是，可避免出现具有不实用名称的传统存取器函数，如 getPropertyName() 和 setPropertyName()。getter 和 setter 的另一个好处是，使用它们可避免允许进行读写访问的每个属性有两个面向公共的函数。

下面的示例类名为 GetSet，其中包含名为 publicAccess() 的 get 和 set 存取器函数，用于提供对名为 privateProperty 的私有变量的访问：

```
class GetSet
{
  private var privateProperty:String;

  public function get publicAccess():String
  {
    return privateProperty;
  }

  public function set publicAccess(setValue:String):void
  {
    privateProperty = setValue;
  }
}
```

如果尝试直接访问属性 privateProperty，将会发生错误，如下所示：

```
var myGetSet:GetSet = new GetSet();
trace(myGetSet.privateProperty); // 发生错误
```

GetSet 类的用户所使用的对象显示为名为 publicAccess 的属性，但实际上这是对名为 privateProperty 的 private 属性执行的一对 get 和 set 存取器函数。下面的示例将实例化 GetSet 类，然后使用名为 publicAccess 的公共存取器设置 privateProperty 的值：

```
var myGetSet:GetSet = new GetSet();
trace(myGetSet.publicAccess); // 输出: null
myGetSet.publicAccess = "hello";
trace(myGetSet.publicAccess); // 输出: hello
```

使用 getter 和 setter 函数还可以覆盖从超类继承来的属性，这是使用常规类成员变量时不能做到的。在子类中不能覆盖使用 var 关键字声明的类成员变量。但是，使用 getter 和 setter 函数创建的属性没有此限制。可以对从超类继承的 getter 和 setter 函数使用 override 属性。

## 绑定方法

绑定方法有时也叫做 "闭包方法"，就是从它的实例提取的方法。作为参数传递给函数的方法或作为值从函数返回的方法都是绑定方法。在 ActionScript 3.0 中，新增的绑定方法类似于闭包函数，其中保留了词汇环境，即使从其实例中提取出来也是如此。绑定方法与闭包函数之间的主要不同差别是，绑定函数的 this 引用保留到实现方法的实例的链接或绑定。换句话说，绑定方法中的 this 引用总是指向实现方法的原始对象。对于闭包函数，this 引用是通用的，这意味着调用函数时，该引用指向与函数关联的任何对象。

如果使用 this 关键字，了解绑定方法就很重要。重新调用 this 关键字可提供对方法父对象的引用。大多数 ActionScript 程序员都希望 this 关键字总是引用包含方法定义的对象或类。但是，如果不使用方法绑定，并不是总是做到这样。例如，在以前版本的 ActionScript 中，this 引用并不总是引用实现方法的实例。从 ActionScript 2.0 的实例中提取方法后，不但 this 引用不绑定到原始实例，而且实例类的成员变量和方法也不可用。在 ActionScript 3.0 中不存在这样的问题，这是因为将方法当作参数传递时会自动创建绑定方法。绑定方法用于确保 this 关键字总是引用在其中定义了方法的对象或类。

下面的代码定义了名为 ThisTest 的类，该类包含一个名为 foo() 的方法（该方法定义绑定方法）和一个名为 bar() 的方法（该方法返回绑定方法）。类外部的代码创建 ThisTest 类的实例，然后调用 bar() 方法，最后将返回值存储在名为 myFunc 的变量中。

```
class ThisTest
{
  private var num:Number = 3;
  function foo():void // 定义的绑定方法
  {
    trace("foo's this: " + this);
    trace("num: " + num);
  }
  function bar():Function
  {
    return foo; // 返回的绑定方法
  }
}
```

```
var myTest:ThisTest = new ThisTest();
var myFunc:Function = myTest.bar();
trace(this); // 输出：[ 全局对象 ]
myFunc();
/* 输出：
foo's this: [object ThisTest]
output: num: 3 */
```

代码的最后两行表明：虽然前一行中的 this 引用指向全局对象，但绑定方法 foo() 中的 this 引用仍然指向 ThisTest 类的实例。另外，存储在 myFunc 变量中的绑定方法仍然可以访问 ThisTest 类的成员变量。如果以上代码在 ActionScript 2.0 中运行，this 引用会匹配，但 num 变量将为 undefined。

绑定方法最值得注意的一种情况是使用事件处理函数，因为 addEventListener() 方法要求将函数或方法作为参数来传递。有关详细信息，请参阅第 281 页的 "定义为类方法的侦听器函数"。

## 类的枚举

"枚举"是您创建的一些自定义数据类型，用于封装一小组值。ActionScript 3.0 并不支持具体的枚举工具，这与 C++ 使用 enum 关键字或 Java 使用 Enumeration 接口不一样。不过，您可以使用类或静态常量创建枚举。例如，Flash Player API 中的 PrintJob 使用名为 PrintJobOrientation 的枚举来存储由 "landscape" 和 "portrait" 组成的一组值，如下面的代码所示：

```
public final class PrintJobOrientation
{
  public static const LANDSCAPE:String = "landscape";
  public static const PORTRAIT:String = "portrait";
}
```

按照惯例，枚举类是使用 final 属性声明的，因为不需要扩展该类。该类仅由静态成员组成，这表示不创建该类的实例。而是直接通过类对象来访问枚举值，如以下代码摘录中所示：

```
var pj:PrintJob = new PrintJob();
if(pj.start())
{
  if (pj.orientation == PrintJobOrientation.PORTRAIT)
  {
    ...
  }
  ...
}
```

Flash Player API 中的所有枚举类都只包含 String、int 或 uint 类型的变量。使用枚举而不使用文本字符串或数字值的好处是，使用枚举更易于发现字面错误。如果枚举名输入错误，ActionScript 编译器会生成一个错误。如果使用字面值，存在拼写错误或使用了错误数字时，编译器并不会报错。在上一个示例中，如果枚举常量的名称不正确，编译器会生成错误，如以下代码摘录中所示：

```
if (pj.orientation == PrintJobOrientation.PORTRAI) // 编译器错误。
```

但是，如果拼错了字符串字面值，编译器并不生成错误，如下所示：

```
if (pj.orientation == "portrai") // 无编译器错误
```

创建枚举的第二种方法还包括使用枚举的静态属性创建单独的类。这种方法的不同之处在于每一个静态属性都包含一个类实例，而不是字符串或整数值。例如，以下代码为一星期中的各天创建了一个枚举类：

```
public final class Day
{
  public static const MONDAY:Day = new Day();
  public static const TUESDAY:Day = new Day();
  public static const WEDNESDAY:Day = new Day();
  public static const THURSDAY:Day = new Day();
  public static const FRIDAY:Day = new Day();
  public static const SATURDAY:Day = new Day();
  public static const SUNDAY:Day = new Day();
}
```

Flash Player API 并不使用这种方法，但是许多开发人员都使用，他们更喜欢使用这种方法提供的改进类型检查功能。例如，返回枚举值的方法可将返回值限定为枚举数据类型。以下代码不但显示了返回星期中各天的函数，还显示了将枚举类型用作类型注释的函数调用：

```
function getDay():Day
{
  var date:Date = new Date();
  var retDay:Day;
  switch (date.day)
  {
    case 0:
      retDay = Day.MONDAY;
      break;
    case 1:
      retDay = Day.TUESDAY;
      break;
    case 2:
      retDay = Day.WEDNESDAY;
      break;
    case 3:
      retDay = Day.THURSDAY;
      break;
    case 4:
      retDay = Day.FRIDAY;
```

```
      break;
    case 5:
      retDay = Day.SATURDAY;
      break;
    case 6:
      retDay = Day.SUNDAY;
      break;
  }
  return retDay;
}

var dayOfWeek:Day = getDay();
```

您还可以增强 Day 类的功能，以使其将一个整数与星期中的各天关联，并提供一个 `toString()` 方法来返回各天的字符串表示形式。您可能希望实践一下，采用这种方法来增强 Day 类的功能。

## 嵌入资源类

ActionScript 3.0 使用称为 "嵌入资源类"的特殊类来表示嵌入的资源。"嵌入资源"指的编译时包括在 SWF 文件中的资源，如声音、图像或字体。嵌入资源而不是动态加载资源，可以确保资源在运行时可用，但代价是增加了 SWF 文件的大小。

### 在 Flash 中使用嵌入资源类

要嵌入资源，首先将该资源放入 FLA 文件的库中。接着，使用资源的链接属性，提供资源的嵌入资源类的名称。如果无法在类路径中找到具有该名称的类，则将自动生成一个类。然后，可以创建嵌入资源类的实例，并使用任何由该类定义或继承的属性和方法。例如，以下代码可用于播放链接到名为 PianoMusic 的嵌入资源类的嵌入声音：

```
var piano:PianoMusic = new PianoMusic();
var sndChannel:SoundChannel = piano.play();
```

# 接口

接口是方法声明的集合，以使不相关的对象能够彼此通信。例如， Flash Player API 定义了 IEventDispatcher 接口，其中包含的方法声明可供类用于处理事件对象。IEventDispatcher 接口建立了标准方法，供对象相互传递事件对象。以下代码显示 IEventDispatcher 接口的定义：

```
public interface IEventDispatcher
{
  function addEventListener(type:String, listener:Function,
      useCapture:Boolean=false, priority:int=0,
      useWeakReference:Boolean = false):void;
  function removeEventListener(type:String, listener:Function,
```

```
        useCapture:Boolean=false):void;
    function dispatchEvent(event:Event):Boolean;
    function hasEventListener(type:String):Boolean;
    function willTrigger(type:String):Boolean;
}
```

接口的基础是方法的接口与方法的实现之间的区别。方法的接口包括调用该方法必需的所有信息，包括方法名、所有参数和返回类型。方法的实现不仅包括接口信息，而且还包括执行方法的行为的可执行语句。接口定义只包含方法接口，实现接口的所有类负责定义方法实现。

在 Flash Player API 中，EventDispatcher 类通过定义所有 IeventDispatcher 接口方法并在每个方法中添加方法体来实现 IEventDispatcher 接口。以下代码摘录自 EventDispatcher 类定义：

```
public class EventDispatcher implements IEventDispatcher
{
    function dispatchEvent(event:Event):Boolean
    {
        /* 实现语句 */
    }

    ...
}
```

IEventDispatcher 接口用作一个协议，EventDispatcher 实例通过该协议处理事件对象，然后将事件对象传递到也实现了 IeventDispatcher 接口的其它对象。

另一种描述接口的方法是：接口定义了数据类型，就像类一样。因此，接口可以用作类型注释，也像类一样。作为数据类型，接口还可以与需要指定数据类型的运算符一起使用，如 is 和 as 运算符。但是与类不同的是，接口不可以实例化。这个区别使很多程序员认为接口是抽象的数据类型，认为类是具体的数据类型。

## 定义接口

接口定义的结构类似于类定义的结构，只是接口只能包含方法但不能包含方法体。接口不能包含变量或常量，但是可以包含 getter 和 setter。要定义接口，请使用 interface 关键字。例如，下面的接口 IExternalizable 是 Flash Player API 中 flash.utils 包的一部分。 IExternalizable 接口定义一个用于对对象进行序列化的协议，这表示将对象转换为适合在设备上存储或通过网络传输的格式。

```
public interface IExternalizable
{
    function writeExternal(output:IDataOutput):void;
    function readExternal(input:IDataInput):void;
}
```

注意，IExternalizable 接口是使用 `public` 访问控制修饰符声明的。只能使用 `public` 和 `internal` 访问控制说明符来修饰接口定义。接口定义中的方法声明不能包含任何访问控制说明符。

Flash Player API 遵循一种约定，其中接口名以大写 `I` 开始，但是可以使用任何合法的标识符作为接口名。接口定义经常位于包的顶级。接口定义不能放在类定义或另一个接口定义中。

接口可扩展一个或多个其它接口。例如，下面的接口 IExample 扩展了 IExternalizable 接口:

```
public interface IExample extends IExternalizable
{
  function extra():void;
}
```

实现 IExample 接口的所有类不但必须包括 `extra()` 方法的实现，还要包括从 Iexternalizable 接口继承的 `writeExternal()` 和 `readExternal()` 方法的实现。

## 在类中实现接口

类是唯一可实现接口的 ActionScript 3.0 语言元素。在类声明中使用 `implements` 关键字可实现一个或多个接口。下面的示例定义了两个接口 IAlpha 和 IBeta 以及实现这两个接口的类 Alpha:

```
interface IAlpha
{
  function foo(str:String):String;
}

interface IBeta
{
  function bar():void;
}

class Alpha implements IAlpha, IBeta
{
  public function foo(param:String):String {}
  public function bar():void {}
}
```

在实现接口的类中，实现的方法必须:

- 使用 `public` 访问控制标识符。
- 使用与接口方法相同的名称。
- 拥有相同数量的参数，每一个参数的数据类型都要与接口方法参数的数据类型相匹配。
- 使用相同的返回类型。

不过在命名所实现方法的参数时，您有一定的灵活性。虽然实现的方法的参数数和每个参数的数据类型必须与接口方法的参数数和数据类型相匹配，但参数名不需要匹配。例如，在上一个示例中，将 Alpha.foo() 方法的参数命名为 param：

```
public function foo(param:String):String {}
```

但是，将 IAlpha.foo() 接口方法中的参数命名为 str：

```
function foo(str:String):String;
```

另外，使用默认参数值也具有一定的灵活性。接口定义可以包含使用默认参数值的函数声明。实现这种函数声明的方法必须采用默认参数值，默认参数值是与接口定义中指定的值具有相同数据类型的一个成员，但是实际值不一定匹配。例如，以下代码定义的接口包含一个使用默认参数值 3 的方法：

```
interface IGamma
{
  function doSomething(param:int = 3):void;
}
```

以下类定义实现 Igamma 接口，但使用不同的默认参数值：

```
class Gamma implements IGamma
{
  public function doSomething(param:int = 4):void {}
}
```

提供这种灵活性的原因是，实现接口的规则的设计目的是确保数据类型兼容性，因此不必要求采用相同的参数名和默认参数名，就能实现目标。

# 继承

继承是指一种代码重用的形式，允许程序员基于现有类开发新类。现有类通常称为 “基类” 或 “超类”，新类通常称为 “子类”。继承的主要优势是，允许重复使用基类中的代码，但不修改现有代码。此外，继承不要求改变其它类与基类交互的方式。不必修改可能已经过彻底测试或可能已被使用的现有类，使用继承可将该类视为一个集成模块，可使用其它属性或方法对它进行扩展。因此，您使用 extends 关键字指明类从另一类继承。

通过继承还可以在代码中利用“多态”。有一种方法在应用于不同数据类型时会有不同行为，多态就是对这样的方法应用一个方法名的能力。名为 Shape 的基类就是一个简单的示例，该类有名为 Circle 和 Square 的两个子类。Shape 类定义了名为 area() 的方法，该方法返回形状的面积。如果已实现多态，则可以对 Circle 和 Square 类型的对象调用 area() 方法，然后执行正确的计算。使用继承能实现多态，实现的方式是允许子类继承和重新定义或“覆盖”基类中的方法。在下面的示例中，由 Circle 和 Square 两个类重新定义了 area() 方法：

```
class Shape
{
  public function area():Number
```

```
    {
        return NaN;
    }
}

class Circle extends Shape
{
    private var radius:Number = 1;
    override public function area():Number
    {
        return (Math.PI * (radius * radius));
    }
}

class Square extends Shape
{
    private var side:Number = 1;
    override public function area():Number
    {
        return (side * side);
    }
}

var cir:Circle = new Circle();
trace(cir.area()); // 输出: 3.141592653589793
var sq:Square = new Square();
trace(sq.area()); // 输出: 1
```

因为每个类定义一个数据类型，所以使用继承会在基类和扩展基类的类之间创建一个特殊关系。子类保证拥有其基类的所有属性，这意味着子类的实例总是可以替换基类的实例。例如，如果方法定义了 Shape 类型的参数 (parameter)，由于 Circle 扩展了 Shape，因此 Circle 类型的参数 (argument) 是合法的，如下所示：

```
function draw(shapeToDraw:Shape) {}

var myCircle:Circle = new Circle();
draw(myCircle);
```

## 实例属性和继承

对于实例属性 (property)，无论是使用 `function`、`var` 还是使用 `const` 关键字定义的，只要在基类中未使用 `private` 属性 (attribute) 声明该属性 (property)，这些属性都可以由子类继承。例如，Flash Player API 中的 Event 类具有很多子类，它们继承了所有事件对象共有的属性。

对于某些类型的事件， Event 类包含了定义事件所需的所有属性。这些类型的事件不需要 Event 类中定义的实例属性以外的实例属性。`complete` 事件和 `connect` 事件就是这样的事件，前者在成功加载数据时发生，后者在建立网络连接时发生。

下面的示例是从 Event 类中摘录的，显示由子类继承的某些属性和方法。由于继承了属性，因此任何子类的实例都可以访问这些属性。

```
public class Event
{
  public function get type():String;
  public function get bubbles():Boolean;
  ...

  public function stopPropagation():void {}
  public function stopImmediatePropagation():void {}
  public function preventDefault():void {}
  public function isDefaultPrevented():Boolean {}
  ...
}
```

其它类型的事件需要 Event 类中没有提供的特有属性。这些事件是使用 Event 类的子类定义的，所以可向 Event 类中定义的属性添加新属性。MouseEvent 类就是这样的一个子类，它可添加与鼠标移动或鼠标单击相关的事件的特有属性，如 mouseMove 和 click 事件。下面的示例是从 MouseEvent 类中摘录的，它说明了在子类中存在但在基类中不存在的属性的定义：

```
public class MouseEvent extends Event
{
  public static const CLICK:String      = "click";
  public static const MOUSE_MOVE:String = "mouseMove";
  ...

  public function get stageX():Number {}
  public function get stageY():Number {}
  ...
}
```

## 访问控制说明符和继承

如果某一属性是用 public 关键字声明的，则该属性对任何位置的代码可见。这表示 public 关键字与 private、protected 和 internal 关键字不同，它对属性继承没有任何限制。

如果属性是使用 private 关键字声明的，该属性只在定义该属性的类中可见，这表示它不能由任何子类继承。此行为与以前版本的 ActionScript 不同，在这些版本中，private 关键字的行为更类似于 ActionScript 3.0 中的 protected 关键字。

protected 关键字指出某一属性不仅在定义该属性的类中可见，而且还在所有子类中可见。与 Java 编程语言中的 protected 关键字不一样，ActionScript 3.0 中的 protected 关键字并不使属性对同一包中的所有其它类可见。在 ActionScript 3.0 中，只有子类可以访问使用 protected 关键字声明的属性。此外， protected 属性对子类可见，不管子类和基类是在同一包中，还是在不同包中。

要限制某一属性在定义该属性的包中的可见性，请使用 internal 关键字或者不使用任何访问控制说明符。未指定访问控制说明符时，应用的默认访问控制说明符是 internal 访问控制说明符。标记为 internal 的属性将只由位于在同一包中的子类继承。

可以使用下面的示例来查看每一个访问控制说明符如何影响跨越包边界的继承。以下代码定义了一个名为 AccessControl 的主应用程序类和两个名为 Base 的 Extender 的其它类。Base 类在名为 foo 的包中，Extender 类是 Base 类的子类，它在名为 bar 的包中。AccessControl 类只导入 Extender 类，并创建一个 Extender 实例，该实例试图访问在 Base 类中定义的名为 str 的变量。 str 变量将声明为 public 以使代码能够进行编译和运行，如以下摘录中所示：

```
// Base.as 位于名为 foo 的文件夹中
package foo
{
  public class Base
  {
    public var str:String = "hello"; // 在该行上更改 public
  }
}

// Extender.as 位于名为 bar 的文件夹中
package bar
{
  import foo.Base;
  public class Extender extends Base
  {
    public function getString():String {
      return str;
    }
  }
}

// 名为 ProtectedExample.as 文件中的主应用程序类
  import flash.display.MovieClip;
  import bar.Extender;
  public class AccessControl extends MovieClip
  {
    public function AccessControl()
    {
      var myExt:Extender = new Extender();
      trace(myExt.testString);  // 如果 str 不是 public，则发生错误
      trace(myExt.getString()); // 如果 str 是 private 或 internal，则发生错误
    }
  }
}
```

要查看其它访问控制说明符如何影响先前示例的编译和执行，请在 AccessControl 类中删除或注释掉以下行，然后将 str 变量的访问控制说明符更改为 private、protected 或 internal：

```
trace(myExt.testString);  // 如果 str 不是 public，则发生错误
```

## 不允许覆盖变量

将继承使用 var 或 const 关键字声明的属性，但不能对其进行覆盖。覆盖某一属性就表示在子类中重新定义该属性。唯一可覆盖的属性类型是方法，即使用 function 关键字声明的属性。虽然不能覆盖实例变量，但是通过为实例变量创建 getter 和 setter 方法并覆盖这些方法，可实现类似的功能。有关详细信息，请参阅第 139 页的 "覆盖 getter 和 setter"。

# 覆盖方法

覆盖方法表示重新定义已继承方法的行为。静态方法不能继承，也不能覆盖。但是，实例方法可由子类继承，也可覆盖，只要符合以下两个条件：

- 实例方法在基类中不是使用 final 关键字声明的。当 final 关键字与实例方法一起使用时，该关键字指明程序员的设计目的是要禁止子类覆盖方法。
- 实例方法在基类中不是使用 private 访问控制说明符声明的。如果某个方法在基类中标记为 private，则在子类中定义同名方法时不需要使用 override 关键字，因为基类方法在子类中不可见。

要覆盖符合这些条件的实例方法，子类中的方法定义必须使用 override 关键字，且必须在以下几个方面与方法的超类版本相匹配：

- 覆盖方法必须与基类方法具有相同级别的访问控制。标记为内部的方法与没有访问控制说明符的方法具有相同级别的访问控制。
- 覆盖方法必须与基类方法具有相同的参数数。
- 覆盖方法参数必须与基类方法参数具有相同的数据类型注释。
- 覆盖方法必须与基类方法具有相同的返回类型。

但是，覆盖方法中的参数名不必与基类中的参数名相匹配，只要参数数和每个参数的数据类相匹配即可。

## super 语句

覆盖方法时，程序员经常希望在要覆盖的超类方法的行为上添加行为，而不是完全替换该行为。这需要通过某种机制来允许子类中的方法调用它本身的超类版本。super 语句就提供了这样一种机制，其中包含对直接超类的引用。下面的示例定义了名为 Base 的类 （其中包含名为 thanks() 的方法），还包含名为 Extender 的 Base 类的子类 （用于覆盖 thanks() 方法）。Extender.thanks() 方法使用 super 语句调用 Base.thanks()。

```
package {
  import flash.display.MovieClip;
  public class SuperExample extends MovieClip
  {
    public function SuperExample()
    {
      var myExt:Extender = new Extender()
      trace(myExt.thanks()); // 输出: Mahalo nui loa
    }
  }
}

class Base {
  public function thanks():String
  {
    return "Mahalo";
  }
}

class Extender extends Base
{
  override public function thanks():String
  {
    return super.thanks() + " nui loa";
  }
}
```

## 覆盖 getter 和 setter

虽然不能覆盖超类中定义的变量，但是可以覆盖 getter 和 setter。例如，以下代码用于覆盖在 Flash Player API 的 MovieClip 类中定义的名为 `currentLabel` 的 getter：

```
package
{
  import flash.display.MovieClip;
  public class OverrideExample extends MovieClip
  {
    public function OverrideExample()
    {
      trace(currentLabel)
    }
    override public function get currentLabel():String
    {
      var str:String = "Override: ";
      str += super.currentLabel;
      return str;
    }
  }
}
```

OverrideExample 类构造函数中的 `trace()` 语句的输出是 `Override: null`，这说明该示例能够覆盖继承的 `currentLabel` 属性。

## 不继承静态属性

静态属性不由子类继承。这意味着不能通过子类的实例访问静态属性。只能通过定义静态属性的类对象来访问静态属性。例如，以下代码定义了名为 Base 的基类和扩展名为 Extender 的 Base 子类。 Base 类中定义了名为 `test` 的静态变量。以下摘录中编写的代码在严格模式下不会进行编译，在标准模式下会生成运行时错误。

```
package {
  import flash.display.MovieClip;
  public class StaticExample extends MovieClip
  {
    public function StaticExample()
    {
      var myExt:Extender = new Extender();
      trace(myExt.test);  // 错误
    }
  }
}

class Base {
  public static var test:String = "static";
}

class Extender extends Base { }
```

访问静态变量 `test` 的唯一方式是通过类对象，如以下代码所示：

```
Base.test;
```

但允许使用与静态属性相同的名称定义实例属性。可以在与静态属性相同的类中或在子类中定义这样的实例属性。例如，以上示例中的 Base 类可以具有一个名为 `test` 的实例属性。在以下代码中，由于 Extender 类继承了实例属性，因此代码能够进行编译和执行。如果 test 实例变量的定义移到 （不是复制） Extender 类，该代码也会编译并执行。

```
package
{
  import flash.display.MovieClip;
  public class StaticExample extends MovieClip
  {
    public function StaticExample()
    {
      var myExt:Extender = new Extender();
      trace(myExt.test);  // 输出：实例
    }
  }
}
```

```
class Base
{
  public static var test:String = "static";
  public var test:String = "instance";
}

class Extender extends Base {}
```

## 静态属性和作用域链

虽然并不继承静态属性，但是静态属性在定义它们的类或该类的任何子类的作用域链中。同样，可以认为静态属性在定义它们的类和任何子类的 “作用域” 中。这意味着在定义静态属性的类体及该类的任何子类中可直接访问静态属性。

下面的示例对上一个示例中定义的类进行了修改，以说明 Base 类中定义的 test 静态变量在 Extender 类的作用域中。换句话说，Extender 类可以访问 test 静态变量，而不必用定义 test 的类名作为变量的前缀。

```
package
{
  import flash.display.MovieClip;
  public class StaticExample extends MovieClip
  {
    public function StaticExample()
    {
      var myExt:Extender = new Extender();
    }
  }
}

class Base {
  public static var test:String = "static";
}

class Extender extends Base
{
  public function Extender()
  {
    trace(test); // 输出: 静态
  }

}
```

如果使用与同类或超类中的静态属性相同的名称定义实例属性，则实例属性在作用域链中的优先级比较高。因此认为实例属性 “遮蔽” 了静态属性，这意味着会使用实例属性的值，而不使用静态属性的值。例如，以下代码显示如果 Extender 类定义名为 test 的实例变量，trace() 语句将使用实例变量的值，而不使用静态变量的值：

```
package
{
  import flash.display.MovieClip;
  public class StaticExample extends MovieClip
  {
    public function StaticExample()
    {
      var myExt:Extender = new Extender();
    }
  }
}

class Base
{
  public static var test:String = "static";
}

class Extender extends Base
{
  public var test:String = "instance";
  public function Extender()
  {
    trace(test); // 输出：实例
  }

}
```

# 高级主题

本节开始先简单介绍 ActionScript 和 OOP 的历史，然后讨论 ActionScript 3.0 对象模型，以及该模型如何启用新的 ActionScript 虚拟机 (AVM2) 显著提供运行速度（与包含旧 ActionScript 虚拟机 (AVM1) 的以前版本的 Flash Player 相比）。

## ActionScript OOP 支持的历史

由于 ActionScript 3.0 是在以前版本的 ActionScript 基础上构建的，了解 ActionScript 对象模型的发展过程可能有所帮助。ActionScript 最初作为早期版本的 Flash 创作工具的简单编写脚本机制。后来，程序员开始使用 ActionScript 建立更加复杂的应用程序。为了迎合这些程序员的需要，每个后续版本都添加了一些语言功能以帮助创建复杂的应用程序。

## ActionScript 1.0

ActionScript 1.0 指在 Flash Player 6 和更早版本中使用的语言版本。即使在这个早期开发阶段，ActionScript 对象模型也是建立在基础数据类型对象的概念的基础上。ActionScript 对象是由一组 “属性” 构成的复合数据类型。讨论对象模型时，术语 “属性” 包括附加到对象的所有内容，如变量、函数或方法。

尽管第一代 ActionScript 不支持使用 `class` 关键字定义类，但是可以使用称为原型对象的特殊对象来定义类。Java 和 C++ 等基于类的语言中使用 `class` 关键字创建要实例化为具体对象的抽象类定义，而 ActionScript 1.0 等基于原型的语言则将现有对象用作其它对象的模型 （或原型）。基于类的语言中的对象可能指向作为其模板的类，而基于原型的语言中的对象则指向作为其模板的另一个对象 （即其原型）。

要在 ActionScript 1.0 中创建类，可以为该类定义一个构造函数。在 ActionScript 中，函数不只是抽象定义，还是实际对象。您创建的构造函数用作该类实例的原型对象。以下代码创建了一个名为 Shape 的类，还定义了一个名为 `visible` 的属性，该属性默认情况下设置为 `true`：

```
// 基类
function Shape() {}
// 创建名为 visible 的属性。
Shape.prototype.visible = true;
```

此构造函数定义了可以使用 `new` 运算符实例化的 Shape 类，如下所示：

```
myShape = new Shape();
```

就像 `Shape()` 构造函数对象用作 Shape 类实例的原型一样，它还可以用作 Shape 的子类 （即扩展 Shape 类的其它类）的原型。

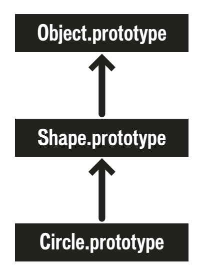
创建作为 Shape 类的子类的类的过程分两个步骤。首先，通过定义类的构造函数创建该类，如下所示：

```
// 子类
function Circle(id, radius)
{
    this.id = id;
    this.radius = radius;
}
```

然后，使用 `new` 运算符将 Shape 类声明为 Circle 类的原型。默认情况下，创建的所有类都使用 Object 类作为其原型，这意味着 `Circle.prototype` 当前包含一个通用对象 （Object 类的实例）。要指定 Circle 的原型是 Shape 而不是 Object，请使用以下代码更改 `Circle.prototype` 的值，使其包含 Shape 对象而不是通用对象。

```
// 使 Circle 成为 Shape 的子类。
Circle.prototype = new Shape();
```

Shape 类和 Circle 类现在通过通常所说的 “原型链”的继承关系联系在一起。下图说明了原型链中的关系：




每个原型链末端的基类是 Object 类。Object 类包含一个名为 `Object.prototype` 的静态属性，该属性指向在 ActionScript 1.0 中创建的所有对象的基础原型对象。示例原型链中的下一个对象是 Shape 对象。这是因为从不显式设置 `Shape.prototype` 属性，所以它仍然包含通用对象 （Object 类的实例） 。此链中的最后一个链环是 Circle 类，该类链接到其原型 Shape 类 （`Circle.prototype` 属性包含 Shape 对象）。

如果创建了 Circle 类的实例 （如下面的示例所示），该实例会继承 Circle 类的原型链：

```
// 创建 Circle 类的实例。
myCircle = new Circle();
```

回想一下，我们创建了一个名为 `visible` 的属性作为 Shape 类的成员。在示例中，`visible` 属性并不作为 `myCircle` 对象的一部分，只是 Shape 对象的一个成员，但以下代码行的输出为 `true`：

```
trace(myCircle.visible); // 输出：true
```

Flash Player 能够沿着原型链检查 `myCircle` 对象是否继承了 `visible` 属性。执行此代码时， Flash Player 首先在 `myCircle` 对象的属性中搜索名为 `visible` 的属性，但是未发现这样的属性。 Flash Player 然后在下一个 `Circle.prototype` 对象中查找，但是仍未发现名为 `visible` 的属性。继续检查原型链，Flash Player 最终发现了在 `Shape.prototype` 对象上定义的 `visible` 属性，并输出该属性的值。

为了简便，本节省略了原型链的很多难懂之处和细节，目的是为了提供足够的信息帮助您了解 ActionScript 3.0 对象模型。

## ActionScript 2.0

在 ActionScript 2.0 中引入了 `class`、 `extends`、 `public` 和 `private` 等新关键字，通过使用这些关键字，您可以按 Java 和 C++ 等基于类的语言用户所熟悉的方式来定义类。ActionScript 1.0 与 ActionScript 2.0 之间的基本继承机制并没有改变，了解这一点很重要。ActionScript 2.0 中只是添加了用于定义类的新语法。在该语言的两个版本中，原型链的作用方式是一样的。

ActionScript 2.0 中引入了新语法（如以下摘录中所示），可允许以数程序员认为更直观的方式定义类：

```
// 基类
class Shape
{
    var visible:Boolean = true;
}
```

注意，ActionScript 2.0 还引入了用于编译时类型检查的类型注释。使用类型注释，可以将上个示例中的 `visible` 属性声明为应只包含布尔值。新 `extends` 关键字还简化了创建子类的过程。在下面的示例中，通过使用 `extends` 关键字，可以一步完成在 ActionScript 1.0 中需要分两步完成的过程：

```
// 子类
class Circle extends Shape
{
  var id:Number;
  var radius:Number;
  function Circle(id, radius)
  {
    this.id = id;
    this.radius = radius;
  }
}
```

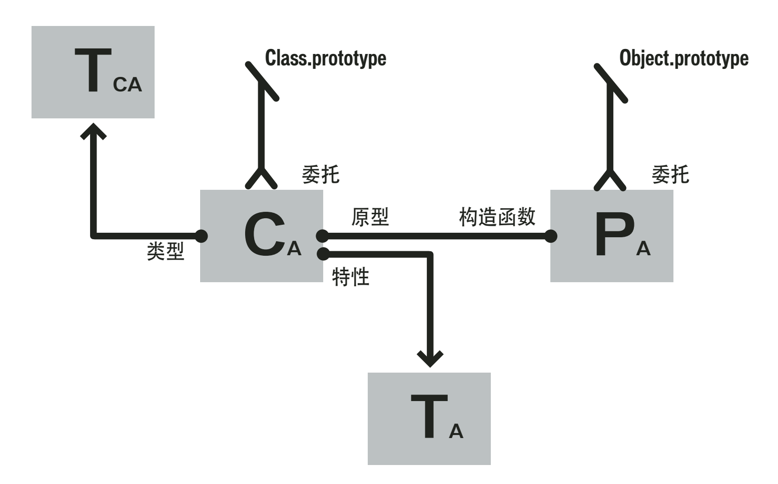
构造函数现在声明为类定义的一部分，还必须显式声明类属性 `id` 和 `radius`。

ActionScript 2.0 中还增加了对接口定义的支持，以使您能够使用为对象间通信正式定义的协议来进一步改进面向对象的程序。

## ActionScript 3.0 类对象

常见的面向对象的编程范例多数常与 Java 和 C++ 相关联，这种范例使用类定义对象的类型。采用这种范例的编程语言也趋向使用类来构造类定义的数据类型的实例。ActionScript 使用类是为了实现以上两个目的，但其根本还是一种基于原型的语言，并带有有趣的特征。ActionScript 为每一类定义创建了特殊的类对象，允许共享行为和状态。但是，对多数 ActionScript 程序员而言，这个特点可能与实际编码没有什么牵连。ActionScript 3.0 的设计目的是，不使用（甚至是不必了解）这些特殊类对象，就可创建复杂的面向对象的 ActionScript 应用程序。对于要利用类对象的高级程序员，本节提供有关问题的深入讨论。

下图显示一个类对象的结构，该类对象表示使用语句 `class A {}` 定义的名为 A 的简单类：




图中的每个矩形表示一个对象。图中的每一个对象都有下标字符 A，这表示该对象属于类 A。类对象 ($C_A$) 包含对许多其它重要对象的引用。实例 traits 对象 ($T_A$) 用于存储在类定义中定义的实例属性。类 traits 对象 ($T_{CA}$) 表示类的内部类型，用于存储该类定义的静态属性（下标字符 C 代表 "类"）。原型对象 ($P_A$) 始终指的是最初通过 `constructor` 属性附加到的类对象。

## traits 对象

traits 对象是 ActionScript 3.0 中的新增对象，它是为了提高性能而实现的。在以前版本的 ActionScript 中，名称查找是一个耗时的过程，因为 Flash Player 要搜索原型链。在 ActionScript 3.0 中，名称查找更有效、耗时更少，因为可以将继承属性从超类复制到子类的 traits 对象。

编程代码不能直接访问 traits 对象，但是性能和内存使用情况的改善可反映它的存在。traits 对象给 AVM2 提供了关于类的布局和内容的详细信息。借助这些信息，AVM2 可显著减少执行时间，因为它可以经常生成直接机器指令来直接访问属性或直接调用方法，而省去了查找名称所耗费的时间。

由于使用了 traits 对象，与以前版本中 ActionScript 类似对象相比，该版本中对象占用内存的时间明显减少。例如，如果某个类已密封（即，该类未声明为 dynamic），则该类实例不需要动态添加属性的哈希表，只保留一个到 traits 对象的指针和该类中定义的固定属性的某些位置。因此，如果对象在 ActionScript 2.0 中需要占用 100 个字节的内存，在 ActionScript 3.0 中只需要占用 20 个字节的内存。

| 注意 | traits 对象是内部实现详细信息，不保证在将来版本的 ActionScript 中此对象不更改，甚至消失。 |
| --- | --- |

## 原型对象

每个 ActionScript 类对象都有一个名为 prototype 属性，它表示对类的原型对象的引用。ActionScript 根本上是基于原型的语言，原型对象是旧内容。有关详细信息，请参阅第 143 页的 “ActionScript 1.0”。

prototype 属性是只读属性，这表示不能将其修改为指向其它对象。这不同于以前版本 ActionScript 中的类 prototype 属性，在以前版本中可以重新分配 prototype，使它指向其它类。虽然 prototype 属性是只读属性，但是它所引用的原型对象不是只读的。换句话说，可以向原型对象添加新属性。向原型对象添加的属性可在类的所有实例中共享。

原型链是以前版本的 ActionScript 中的唯一继承机制，在 ActionScript 3.0 中只充当一个辅助角色。主要的继承机制固定属性继承由 traits 对象内部处理。固定属性指的是定义为类定义的一部分的变量或方法。固定属性继承也叫做类继承，因为它是与 class、extends 和 override 等关键字相关的继承机制。

原型链提供了另一种继承机制，该机制的动态性比固定属性继承的更强。既可以将属性作为类定义的一部分，也可以在运行时通过类对象的 prototype 属性向类的原型对象中添加属性。但是，请注意，如果将编译器设置为严格模式，则不能访问添加到原型对象中的属性，除非使用 dynamic 关键字声明类。

Object 类就是这样类的示例，它的原型对象附加了若干属性。 Object 类的 toString() 和 valueOf() 方法实际上是一些函数，它们分配给 Object 类原型对象的属性。以下是一个示例，说明这些方法的声明理论上是怎样的 （实际实现时会因实现详细信息而稍有不同）：

```
public dynamic class Object
{
    prototype.toString = function()
    {
        // 语句
    };
    prototype.valueOf = function()
    {
        // 语句
    };
}
```

正如前面提到的那样，可以将属性附加到类定义外部的类原型对象。例如，也可以在 Object 类定义外部定义 toString() 方法，如下所示：

```
Object.prototype.toString = function()
{
    // 语句
};
```

但是，原型继承与固定属性继承不一样，如果要重新定义子类中的方法，原型继承不需要 override 关键字。例如，如果要重新定义 Object 类的子类中的 valueOf() 方法，您有以下三种选择。第一，可以在类定义中的子类原型对象上定义 valueOf() 方法。以下代码创建一个名为 Foo 的 Object 子类，还将 Foo 原型对象的 valueOf() 方法重新定义为类定义的一部分。因为每个类都是从 Object 继承的，所以不需要使用 extends 关键字。

```
dynamic class Foo
{
  prototype.valueOf = function()
  {
    return "Instance of Foo";
  };
}
```

第二，可以在类定义外部对 Foo 原型对象定义 valueOf() 方法，如以下代码中所示：

```
Foo.prototype.valueOf = function()
{
  return "Instance of Foo";
};
```

第三，可以将名为 valueOf() 的固定属性定义为 Foo 类的一部分。这种方法与其它混合了固定属性继承与原型继承的方法有所不同。要重新定义 valueOf() 的 Foo 的任何子类必须使用 override 关键字。以下代码显示 valueOf() 定义为 Foo 中的固定属性：

```
class Foo
{
  function valueOf():String
  {
    return "Instance of Foo";
  }
}
```

## AS3 命名空间

由于存在两种继承机制，即固定属性继承和原型继承，所以涉及到核心类的属性和方法时，就存在两种机制的兼容性问题。如果与 ECMAScript 第 4 版语言规范草案兼容，则要求使用原型继承，这意味着核心类的属性和方法是在该类的原型对象上定义的。另一方面，如果与 Flash Player API 兼容，则要求使用固定属性继承，这意味着核心类的属性和方法是使用 const、var 和 function 关键字在类定义中定义的。此外，如果使用固定属性而不是原型属性，将显著提升运行时性能。

在 ActionScript 3.0 中，通过同时将原型继承和固定属性继承用于核心类，解决了这个问题。每一个核心类都包含两组属性和方法。一组是在原型对象上定义的，用于与 ECMAScript 规范兼容，另一组使用固定属性定义和 AS3 命名空间定义，以便与 Flash Player API 兼容。

AS3 命名空间提供了一种约定机制，用来在两组属性和方法之间做出选择。如果不使用 AS3 命名空间，核心类的实例会继承在核心类的原型对象上定义的属性和方法。如果决定使用 AS3 命名空间，核心类的实例会继承 AS3 版本，因为固定属性的优先级始终高于原型属性。换句话说，只要固定属性可用，则始终使用固定属性，而不使用同名的原型属性。

通过用 AS3 命名空间限定属性或方法，可以选择使用 AS3 命名空间版本的属性或方法。例如，下面的代码使用 AS3 版本的 `Array.pop()` 方法：

```
var nums:Array = new Array(1, 2, 3);
nums.AS3::pop();
trace(nums); // 输出: 1,2
```

或者，也可以使用 `use namespace` 指令打开代码块中所有定义的 AS3 命名空间。例如，以下代码使用 `use namespace` 指令打开 `pop()` 和 `push()` 方法的 AS3 命名空间：

```
use namespace AS3;

var nums:Array = new Array(1, 2, 3);
nums.pop();
nums.push(5);
trace(nums) // 输出: 1,2,5
```

ActionScript 3.0 还为每组属性提供了编译器选项，以便将 AS3 命名空间应用于整个程序。`-as3` 编译器选项表示 AS3 命名空间，`-es` 编译器选项表示原型继承选项（`es` 代表 ECMAScript）。要打开整个程序的 AS3 命名空间，请将 `-as3` 编译器选项设置为 `true`，将 `-es` 编译器选项设置为 `false`。要使用原型版本，请将编译器选项设置为相反值。Adobe Flex Builder 2 和 Adobe Flash CS3 Professional 的默认编译器选项是 `-as3 = true` 和 `-es = false`。

如果计划扩展任何核心类并覆盖任何方法，应了解 AS3 命名空间对声明覆盖方法的方式有什么影响。如果要使用 AS3 命名空间，覆盖核心类方法的任何方法都必须使用 AS3 命名空间以及 `override` 属性。如果不打算使用 AS3 命名空间且要重新定义子类中的核心类方法，则不应使用 AS3 命名空间或 `override` 关键字。

# 示例：GeometricShapes

GeometricShapes 范例应用程序说明了如何使用 ActionScript 3.0 来应用很多面向对象的概念和功能，其中包括：

- 定义类
- 扩展类
- 多态和 `override` 关键字
- 定义、扩展和实现接口

示例中还包括一个用于创建类实例的 “工厂方法”，说明如何将返回值声明为接口的实例，以及通过一般方法使用返回的对象。

要获取该范例的应用程序文件，请访问
www.adobe.com/go/learn_programmingAS3samples_flash_cn。
在 Samples/GeometricShapes 文件夹下可找到 GeometricShapes 应用程序文件。该应用程序包含以下文件：

| 文件 | 说明 |
|---|---|
| GeometricShapes.mxml<br>或<br>GeometricShapes.fla | Flash 或 Flex 中的主应用程序文件 （分别为 FLA 和 MXML）。 |
| com/example/programmingas3/geometricshapes/IGeometricShape.as | 由所有 GeometricShapes 应用程序类实现的基接口定义方法。 |
| com/example/programmingas3/geometricshapes/IPolygon.as | 由有多条边的 GeometricShapes 应用程序类实现的接口定义方法。 |
| com/example/programmingas3/geometricshapes/RegularPolygon.as | 一种几何形状，这种几何形状的边长相等，并且这些边围绕形状中心对称分布。 |
| com/example/programmingas3/geometricshapes/Circle.as | 一种用于定义圆的几何形状。 |
| com/example/programmingas3/geometricshapes/EquilateralTriangle.as | RegularPolygon 的子类，用于定义所有边长相等的三角形。 |
| com/example/programmingas3/geometricshapes/Square.as | RegularPolygon 的子类，用于定义所有四条边相等的矩形。 |
| com/example/programmingas3/geometricshapes/GeometricShapeFactory.as | 包含工厂方法的一个类，用于创建给定了形状类型和尺寸的形状。 |

## 定义 GeometricShapes 类

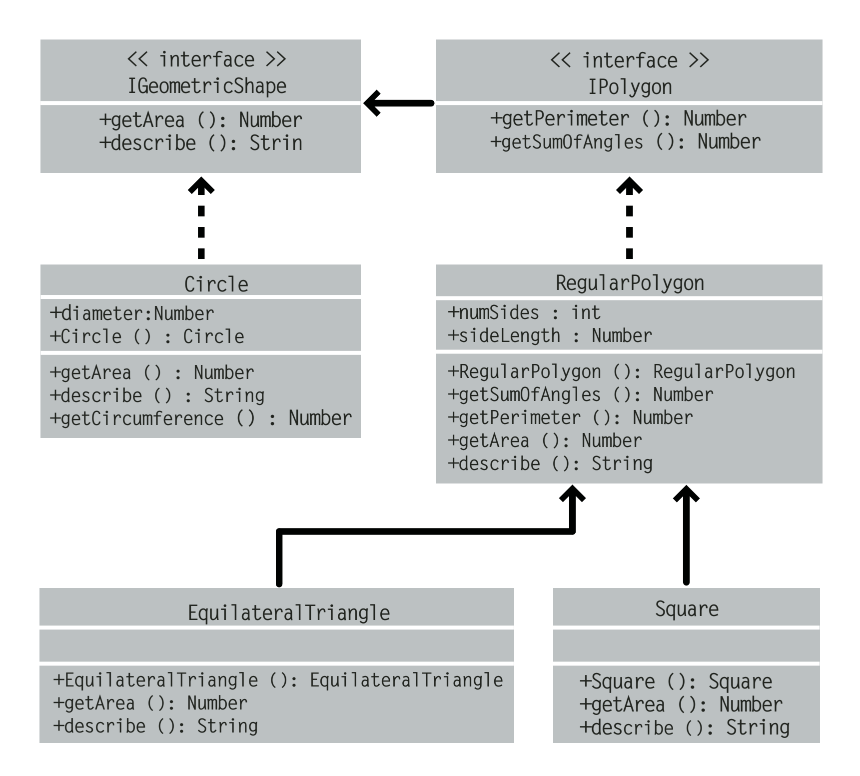
使用 GeometricShapes 应用程序可以允许用户指定几何形状的类型和尺寸。随后，应用程序通过形状描述、其面积和周长进行响应。

应用程序的用户界面显得很零碎，其中包括一些可用于选择形状类型、设置尺寸及显示描述的控件。这个应用程序最引人注意的部分是程序的内部，即在类和接口本身的结构中。

这个应用程序用于处理几何形状，但是它并不以图形化方式显示这些形状。它提供一个由类和接口组成的小型库，后面的章节会重复使用这个库 （请参阅第 363 页的 “示例：SpriteArranger”）。 SpriteArranger 示例以图形化方式显示形状，还允许用户根据 GeometricShapes 应用程序中提供的类框架处理形状。

在下图中使用统一建模语言 (UML) 表示法显示此示例中用于定义几何形状的类和接口：

**GeometricShapes 示例类**




# 定义接口的通用行为

这个 GeometricShapes 应用程序处理三种形状：圆、正方形和等边三角形。GeometricShapes 类结构以非常简单的接口 IGeometricShape 作为开始，它列出通用于所有这三种形状的方法。

```
package com.example.programmingas3.geometricshapes
{
   public interface IGeometricShape
   {
      function getArea():Number;
      function describe():String;
   }
}
```

该接口定义了两个方法：`getArea()` 方法和 `describe()` 方法，前者计算并返回形状的面积，后者汇编形状属性的文本描述。

我们还要知道每个形状的周边长度。但是，圆的周边长度叫做周长，它的计算方式是独有的，所以其行为异于三角形或正方形的行为。不过，三角形、正方形和其它多边形之间仍然有很多类似之处，所以为它们定义一个新接口类 IPolygon 还是很有意义的。IPolygon 接口也相当简单，如下所示：

```
package com.example.programmingas3.geometricshapes
{
  public interface IPolygon extends IGeometricShape
  {
    function getPerimeter():Number;
    function getSumOfAngles():Number;
  }
}
```

这个接口定义了所有多边形的两个通用方法：`getPerimeter()` 方法和 `getSumOfAngles()` 方法，前者用于计算所有边相加后的总长，后者用于将所有内角相加起来。

IPolygon 接口扩展了 IGeometricShape 接口，这意味着实现 IPolygon 接口的所有类必须声明所有四个方法，即来自 IGeometricShape 接口的两个方法和来自 IPolygon 接口的两个方法。

## 定义形状类

对每种形状的方法有所了解后，就可以定义形状类本身了。就需要实现的方法数而言，最简单的形状是 Circle 类，如下所示：

```
package com.example.programmingas3.geometricshapes
{
  public class Circle implements IGeometricShape
  {
    public var diameter:Number;

    public function Circle(diam:Number = 100):void
    {
      this.diameter = diam;
    }

    public function getArea():Number
    {
      // The formula is Pi * radius * radius.
      var radius:Number = diameter / 2;
      return Math.PI * radius * radius;
    }

    public function getCircumference():Number
    {
      // The formula is Pi * diameter.
      return Math.PI * diameter;
    }
```

```
    public function describe():String
    {
      var desc:String = "This shape is a Circle.\n";
      desc += "Its diameter is " + diameter + " pixels.\n";
      desc += "Its area is " + getArea() + ".\n";
      desc += "Its circumference is " + getCircumference() + ".\n";
      return desc;
    }
  }
}
```

Circle 类用于实现 IGeometricShape 接口，因此，它必须为 `getArea()` 方法和 `describe()` 方法提供代码。此外，它还定义 `getCircumference()` 方法，该方法对 Circle 类是唯一的。另外， Circle 类还声明属性 `diameter`，在其它多边形类中找不到该属性。

其它两种形状，即正方形和等边三角形的某些其它方面是共同的：它们各自的边长相等，各自都有可用于计算周长和内角总和的通用公式。实际上，这些通用公式还适用于将来需要定义的任何其它正多边形。

RegularPolygon 类将是 Square 和 EquilateralTriangle 两个类的超类。使用超类可在一个位置定义通用方法，所以不必在每个子类中单独定义方法。以下是 RegularPolygon 类的代码：

```
package com.example.programmingas3.geometricshapes
{
  public class RegularPolygon implements IPolygon
  {
    public var numSides:int;
    public var sideLength:Number;

    public function RegularPolygon(len:Number = 100, sides:int = 3):void
    {
      this.sideLength = len;
      this.numSides = sides;
    }

    public function getArea():Number
    {
      // This method should be overridden in subclasses.
      return 0;
    }

    public function getPerimeter():Number
    {
      return sideLength * numSides;
    }

    public function getSumOfAngles():Number
    {
      if (numSides >= 3)
      {
        return ((numSides - 2) * 180);
```

```
      }
      else
      {
        return 0;
      }
    }

    public function describe():String
    {
      var desc:String = "Each side is " + sideLength + " pixels long.\n";
      desc += "Its area is " + getArea() + " pixels square.\n";
      desc += "Its perimeter is " + getPerimeter() + " pixels long.\n";
      desc += "The sum of all interior angles in this shape is " +
  getSumOfAngles() + " degrees.\n";
      return desc;
    }
  }
}
```

第一，RegularPolygon 类用于声明所有正多边形的两个通用属性: 每边的长度 (`sideLength` 属性) 和边数 (`numSides` 属性)。

RegularPolygon 类实现 IPolygon 接口，还声明 IPolygon 接口的所有四个方法。它使用通用公式来实现 `getPerimeter()` 和 `getSumOfAngles()` 两种方法。

因为用于 `getArea()` 方法的公式将因形状的不同而有所不同，所以该方法的基类版本不能包括可由子类方法继承的通用逻辑。而是只返回 0 默认值来指明未计算面积。要正确地计算每一个形状的面积， RegularPolygon 类的子类本身一定要覆盖 `getArea()` 方法本身。

EquilateralTriangle 类的以下代码说明如何覆盖 `getArea()` 方法:

```
package com.example.programmingas3.geometricshapes
{
  public class EquilateralTriangle extends RegularPolygon
  {
    public function EquilateralTriangle(len:Number = 100):void
    {
      super(len, 3);
    }

    public override function getArea():Number
    {
        // The formula is ((sideLength squared) * (square root of 3)) / 4.
        return ( (this.sideLength * this.sideLength) * Math.sqrt(3) ) / 4;
    }

    public override function describe():String
    {
         /* starts with the name of the shape, then delegates the rest
         of the description work to the RegularPolygon superclass */
        var desc:String = "This shape is an equilateral Triangle.\n";
```

```
            desc += super.describe();
            return desc;
        }
    }
}
```

override 关键字指示 EquilateralTriangle.getArea() 方法有意覆盖 RegularPolygon 超类中的 getArea() 方法。当调用 EquilateralTriangle.getArea() 方法时，该方法使用先前代码中的公式计算面积，从不执行 RegularPolygon.getArea() 方法中的代码。

不同的是， EquilateralTriangle 类并不定义它自己版本的 getPerimeter() 方法。当调用 EquilateralTriangle.getPerimeter() 方法时，调用会沿继承链找到 RegularPolygon 超类的 getPerimeter() 方法，并执行其中的代码。

EquilateralTriangle() 构造函数使用 super() 语句显式地调用其超类的 RegularPolygon() 构造函数。如果这两个构造函数使用同一组参数，则可以完全省略 EquilateralTriangle() 构造函数，而执行 RegularPolygon() 构造函数。但是，RegularPolygon() 构造函数需要额外的参数，即 numSides。所以，EquilateralTriangle() 函数调用 super(len, 3)，该语句传递 len 输入参数和值 3，以指示三角形有 3 个边。

describe() 方法也使用 super() 语句，但是采用不同的方式调用 describe() 方法的 RegularPolygon 超类版本。 EquilateralTriangle.describe() 方法先将 desc 字符串变量设置为有关形状类型的语句。然后调用 super.describe() 来获取 RegularPolygon.describe() 方法的结果，之后将结果追加到 desc 字符串。

此处不详细讨论 Square 类，但它类似于 EquilateralTriangle 类，它们都提供构造函数及其自己的 getArea() 和 describe() 方法实现。

## 多态和工厂方法

可以采用许多有趣的方法正确使用接口的一组类及继承。例如，到目前为止讨论的所有形状类不是用于实现 IGeometricShape 接口，就是用于扩展执行的超类。因此，如果将一个变量定义为 IGeometricShape 实例，您不必知道它实际是 Circle 类的实例还是 Square 类的实例，就可以调用它的 describe() 方法。

以下代码说明这是如何实现的：

```
var myShape:IGeometricShape = new Circle(100);
trace(myShape.describe());
```

当调用 myShape.describe() 时，它执行 Circle.describe() 方法，因为即使将变量定义为 IGeometricShape 接口的实例， Circle 也是它的基础类。

这个示例说明了运行时多态的原则：完全相同的方法调用会导致执行不同的代码，这取决于要调用了其方法的对象的类。

GeometricShapes 应用程序会使用设计模式的简化版本 （叫做 “工厂方法”）应用这种基于接口的多态原则。术语 “工厂方法”指的是一个函数，该函数返回一个对象，其基本数据类型或内容可能会因上下文而不同。

以下显示的 GeometricShapeFactory 类定义一个名为 createShape() 的工厂方法：

```
package com.example.programmingas3.geometricshapes
{
  public class GeometricShapeFactory
  {
    public static var currentShape:IGeometricShape;

    public static function createShape(shapeName:String,
                                        len:Number):IGeometricShape
    {
      switch (shapeName)
      {
        case "Triangle":
          return new EquilateralTriangle(len);

        case "Square":
          return new Square(len);

        case "Circle":
          return new Circle(len);
      }
      return null;
    }

    public static function describeShape(shapeType:String,
  shapeSize:Number):String
    {
      GeometricShapeFactory.currentShape =
        GeometricShapeFactory.createShape(shapeType, shapeSize);
      return GeometricShapeFactory.currentShape.describe();
    }
  }
}
```

createShape() 工厂方法允许形状子类构造函数定义所创建的实例的详细信息，同时返回新对象作为 IGeometricShape 实例，以便应用程序可以采用更通用的方式处理这些对象。

前面示例中的 describeShape() 方法说明应用程序如何使用工厂方法来获取对更具体对象的一般引用。该应用程序可通过以下方式获取新创建的 Circle 对象的描述：

```
GeometricShapeFactory.describeShape(“Circle”, 100);
```

然后，describeShape() 方法使用相同的参数调用 createShape() 工厂方法，以将新的 Circle 对象存储在名为 currentShape 的静态变量 （该变量是作为 IGeometricShape 对象键入的）中。接下来，对 currentShape 调用 describe() 方法，该调用将被自动解析以执行 Circle.describe() 方法，从而返回圆的详细描述。

## 增强范例应用程序

增强或更改应用程序后，接口和继承的实际作用会非常明显。

假定要在这个示例应用程序中添加新形状，即一个五边形。您需要创建一个新 Pentagon 类，以扩展 RegularPolygon 类并定义它自己版本的 `getArea()` 和 `describe()` 方法。然后要在应用程序用户界面的组合框中添加一个新 Pentagon 选项。这就完成了。Pentagon 类将通过继承来自动获取 RegularPolygon 类的 `getPerimeter()` 方法和 `getSumOfAngles()` 方法的功能。由于该类是从实现 IGeometricShape 接口的类继承的，因此 Pentagon 实例也可以视为 IGeometricShape 实例。这意味着不需要更改 GeometricShapeFactory 类中的任何方法，因此在需要时可以很方便地添加新的形状类型。

您可能希望实践一下，将 Pentagon 类添加到 Geometric Shapes 示例，以了解向应用程序中添加新功能时，使用接口和继承将如何减轻工作量。

第 5 章

# 处理日期和时间

5

时间可能并不代表一切，但它在软件应用程序中通常是一个关键要素。ActionScript 3.0 提供了多种强大的手段来管理日历日期、时间和时间间隔。以下两个主类提供了大部分的计时功能：Date 类和 flash.utils 包中的新 Timer 类。

## 目录

# 日期和时间基础知识

## 处理日期和时间简介

日期和时间是在 ActionScript 程序中使用的一种常见信息类型。例如，您可能需要了解当前星期值，或测量用户在特定屏幕上花费多少时间，并且还可能会执行很多其它操作。在 ActionScript 中，可以使用 Date 类来表示某一时刻，其中包含日期和时间信息。 Date 实例中包含各个日期和时间单位的值，其中包括年、月、日、星期、小时、分钟、秒、毫秒以及时区。对于更高级的用法， ActionScript 还包括 Timer 类，您可以使用该类在一定延迟后执行动作，或按重复间隔执行动作。

## 常见日期和时间任务

本章介绍了以下常见的日期和时间信息处理任务：

- 处理 Date 对象
- 获取当前日期和时间
- 访问各个日期和时间单位 （日、年、小时、分钟等）
- 使用日期和时间执行运算
- 在时区之间进行转换
- 执行重复动作
- 在设定的时间间隔后执行动作

## 重要概念和术语

以下参考列表包含将会在本章中遇到的重要术语：

- UTC 时间 (UTC time)：通用协调时间，即 “零小时”基准时区。所有其它时区均被定义为相对于 UTC 时间快或慢一定的小时数。

## 完成本章中的示例

学习本章的过程中，您可能想要自己动手测试一些示例代码清单。由于本章中的代码清单主要用于处理 Date 对象，测试示例将涉及查看示例中使用的变量值，方法是：通过将值写入舞台上的文本字段实例，或使用 `trace()` 函数将值输出到 “输出”面板。第 53 页的 “测试本章内的示例代码清单”中对这些技术进行了详细说明。

# 管理日历日期和时间

ActionScript 3.0 的所有日历日期和时间管理函数都集中在顶级 Date 类中。Date 类包含一些方法和属性，这些方法和属性能够使您按照通用协调时间 (UTC) 或特定于时区的本地时间来处理日期和时间。 UTC 是一种标准时间定义，它实质上与格林尼治标准时间 (GMT) 相同。

## 创建 Date 对象

Date 类是所有核心类中构造函数方法形式最为多变的类之一。您可以用以下四种方式来调用 Date 类。

第一，如果未给定参数，则 `Date()` 构造函数将按照您所在时区的本地时间返回包含当前日期和时间的 Date 对象。下面是一个示例：

```
var now:Date = new Date();
```

第二，如果仅给定了一个数字参数，则 `Date()` 构造函数将其视为自 1970 年 1 月 1 日以来经过的毫秒数，并且返回对应的 Date 对象。请注意，您传入的毫秒值将被视为自 1970 年 1 月 1 日 （UTC 时间）以来经过的毫秒数。但是，该 Date 对象会按照您所在的本地时区来显示值，除非您使用特定于 UTC 的方法来检索和显示这些值。如果仅使用一个毫秒参数来创建新的 Date 对象，则应确保考虑到您的当地时间和 UTC 之间的时区差异。以下语句创建一个设置为 1970 年 1 月 1 日午夜 （UTC 时间）的 Date 对象：

```
var millisecondsPerDay:int = 1000 * 60 * 60 * 24;
// 获取一个表示自起始日期 1970 年 1 月 1 日后又过了一天时间的 Date 对象
var startTime:Date = new Date(millisecondsPerDay);
```

第三，您可以将多个数值参数传递给 `Date()` 构造函数。该构造函数将这些参数分别视为年、月、日、小时、分钟、秒和毫秒，并将返回一个对应的 Date 对象。假定这些输入参数采用的是本地时间而不是 UTC。以下语句获取一个设置为 2000 年 1 月 1 日开始的午夜（本地时间）的 Date 对象：

```
var millenium:Date = new Date(2000, 0, 1, 0, 0, 0, 0);
```

第四，您可以将单个字符串参数传递给 `Date()` 构造函数。该构造函数将尝试把字符串解析为日期或时间部分，然后返回对应的 Date 对象。如果您使用此方法，最好将 `Date()` 构造函数包含在 `try..catch` 块中以捕获任何解析错误。`Date()` 构造函数接受多种不同的字符串格式，如《ActionScript 3.0 语言和组件参考》中所列。以下语句使用字符串值初始化一个新的 Date 对象：

```
var nextDay:Date = new Date("Mon May 1 2006 11:30:00 AM");
```

如果 `Date()` 构造函数无法成功解析该字符串参数，它将不会引发异常。但是，所得到的 Date 对象将包含一个无效的日期值。

## 获取时间单位值

可以使用 Date 类的属性或方法从 Date 对象中提取各种时间单位的值。下面的每个属性为您提供了 Date 对象中的一个时间单位的值：

- `fullYear` 属性
- `month` 属性，以数字格式表示，分别以 0 到 11 表示一月到十二月
- `date` 属性，表示月中某一天的日历数字，范围从 1 到 31
- `day` 属性，以数字格式表示一周中的某一天，其中 0 表示星期日
- `hours` 属性，范围从 0 到 23
- `minutes` 属性
- `seconds` 属性
- `milliseconds` 属性

实际上， Date 类为您提供了获取这些值的多种方式。例如，您可以用四种不同方式获取 Date 对象的月份值：

- `month` 属性
- `getMonth()` 方法
- `monthUTC` 属性
- `getMonthUTC()` 方法

所有四种方式实质上具有同等的效率，因此您可以任意使用一种最适合应用程序的方法。

刚才列出的属性表示总日期值的各个部分。例如，milliseconds 属性永远不会大于 999，因为当它达到 1000 时，秒钟值就会增加 1 并且 milliseconds 属性会重置为 0。

如果要获得 Date 对象自 1970 年 1 月 1 日 (UTC) 起所经过毫秒数的值，您可以使用 `getTime()` 方法。通过使用与其相对应的 `setTime()` 方法，您可以使用自 1970 年 1 月 1 日 (UTC) 起经过的毫秒数更改现有 Date 对象的值。

## 执行日期和时间运算

您可以使用 Date 类对日期和时间执行加法和减法运算。日期值在内部以毫秒的形式保存，因此您应将其它值转换成毫秒，然后再将它们与 Date 对象进行加减。

如果应用程序将执行大量的日期和时间运算，您可能会发现创建常量来保存常见时间单位值（以毫秒的形式）非常有用，如下所示：

```
public static const millisecondsPerMinute:int = 1000 * 60;
public static const millisecondsPerHour:int = 1000 * 60 * 60;
public static const millisecondsPerDay:int = 1000 * 60 * 60 * 24;
```

现在，可以方便地使用标准时间单位来执行日期运算。下列代码使用 getTime() 和 setTime() 方法将日期值设置为当前时间一个小时后的时间：

```
var oneHourFromNow:Date = new Date();
oneHourFromNow.setTime(oneHourFromNow.getTime() + millisecondsPerHour);
```

设置日期值的另一种方式是仅使用一个毫秒参数创建新的 Date 对象。例如，下列代码将一个日期加上 30 天以计算另一个日期：

```
// 将发票日期设置为今天的日期
var invoiceDate:Date = new Date();

// 加上 30 天以获得到期日期
var dueDate:Date = new Date(invoiceDate.getTime() + (30 * millisecondsPerDay));
```

接着，将 millisecondsPerDay 常量乘以 30 以表示 30 天的时间，并将得到的结果与 invoiceDate 值相加并将其用于设置 dueDate 值。

## 在时区之间进行转换

在需要将日期从一种时区转换成另一种时区时，使用日期和时间运算十分方便。也可以使用 getTimezoneOffset() 方法，该方法返回的值表示 Date 对象的时区与 UTC 之间相差的分钟数。此方法之所以返回以分钟为单位的值是因为并不是所有时区之间都正好相差一个小时，有些时区与邻近的时区仅相差半个小时。

以下示例使用时区偏移量将日期从本地时间转换成 UTC。该示例首先以毫秒为单位计算时区值，然后按照该量调整 Date 值：

```
// 按本地时间创建 Date
var nextDay:Date = new Date("Mon May 1 2006 11:30:00 AM");

// 通过加上或减去时区偏移量，将 Date 转换为 UTC
var offsetMilliseconds:Number = nextDay.getTimezoneOffset() * 60 * 1000;
nextDay.setTime(nextDay.getTime() + offsetMilliseconds);
```

# 控制时间间隔

使用 Adobe Flash CS3 Professional 开发应用程序时，您可以访问时间轴，这会使您稳定且逐帧地完成该应用程序。但在纯 ActionScript 项目中，您必须依靠其它计时机制。

## 循环与计时器之比较

在某些编程语言中，您必须使用循环语句（如 `for` 或 `do..while`）来设计您自己的计时方案。

通常，循环语句会以本地计算机所允许的速度尽可能快地执行，这表明应用程序在某些计算机上的运行速度较快而在其它计算机上则较慢。如果应用程序需要一致的计时间隔，则您需要将其与实际的日历或时钟时间联系在一起。许多应用程序（如游戏、动画和实时控制器）需要在不同计算机上均能保持一致的、规则的时间驱动计时机制。

ActionScript 3.0 的 Timer 类提供了一个功能强大的解决方案。使用 ActionScript 3.0 事件模型， Timer 类在每次达到指定的时间间隔时都会调度计时器事件。

# Timer 类

在 ActionScript 3.0 中处理计时函数的首选方式是使用 Timer 类 (flash.utils.Timer)，可以使用它在每次达到间隔时调度事件。

要启动计时器，请先创建 Timer 类的实例，并告诉它每隔多长时间生成一次计时器事件以及在停止前生成多少次事件。

例如，下列代码创建一个每秒调度一个事件且持续 60 秒的 Timer 实例：

```
var oneMinuteTimer:Timer = new Timer(1000, 60);
```

Timer 对象在每次达到指定的间隔时都会调度 TimerEvent 对象。TimerEvent 对象的事件类型是 `timer`（由常量 `TimerEvent.TIMER` 定义）。 TimerEvent 对象包含的属性与标准 Event 对象包含的属性相同。

如果将 Timer 实例设置为固定的间隔数，则在达到最后一次间隔时，它还会调度 `timerComplete` 事件（由常量 `TimerEvent.TIMER_COMPLETE` 定义）。

以下是一个用来展示 Timer 类实际操作的小示例应用程序：

```
package
{
  import flash.display.Sprite;
  import flash.events.TimerEvent;
  import flash.utils.Timer;

  public class ShortTimer extends Sprite
  {
    public function ShortTimer()
    {
```

```
        // 创建一个新的五秒的 Timer
        var minuteTimer:Timer = new Timer(1000, 5);

        // 为间隔和完成事件指定侦听器
        minuteTimer.addEventListener(TimerEvent.TIMER, onTick);
        minuteTimer.addEventListener(TimerEvent.TIMER_COMPLETE,
    onTimerComplete);

        // 启动计时器计时
        minuteTimer.start();
      }

      public function onTick(event:TimerEvent):void
      {
        // 显示到目前为止的时间计数
        // 该事件的目标是 Timer 实例本身。
        trace("tick" + event.target.currentCount);
      }

      public function onTimerComplete(event:TimerEvent):void
      {
        trace("Time's Up!");
      }
    }
}
```

创建 ShortTimer 类时，它会创建一个用于每秒计时一次并持续五秒的 Timer 实例。然后，它将两个侦听器添加到计时器：一个用于侦听每次计时，另一个用于侦听 `timerComplete` 事件。

接着，它启动计数器计时，并且从此时起以一秒钟的间隔执行 `onTick()` 方法。

`onTick()` 方法只显示当前的时间计数。五秒钟后，执行 `onTimerComplete()` 方法，告诉您时间已到。

运行该示例时，您应会看到下列行以每秒一行的速度显示在控制台或跟踪窗口中：

```
tick 1
tick 2
tick 3
tick 4
tick 5
Time's Up!
```

## flash.utils 包中的计时函数

ActionScript 3.0 包含许多与 ActionScript 2.0 提供的计时函数类似的计时函数。这些函数是作为 flash.utils 包中的包级别函数提供的，这些函数的工作方式与 ActionScript 2.0 完全相同。

| 函数 | 说明 |
| --- | --- |
| `clearInterval(id:uint):void` | 取消指定的 `setInterval()` 调用。 |
| `clearTimeout(id:uint):void` | 取消指定的 `setTimeout()` 调用。 |
| `getTimer():int` | 返回自 Adobe Flash Player 被初始化以来经过的毫秒数。 |
| `setInterval(closure:Function, delay:Number, ... arguments):uint` | 以指定的间隔 （以毫秒为单位）运行函数。 |
| `setTimeout(closure:Function, delay:Number, ... arguments):uint` | 在指定的延迟 （以毫秒为单位）后运行指定的函数。 |

这些函数仍保留在 ActionScript 3.0 以实现向后兼容。 Adobe 不建议您在新的 ActionScript 3.0 应用程序中使用这些函数。通常，在应用程序中使用 Timer 类会更容易且更有效。

# 示例：简单的模拟时钟

简单的模拟时钟示例说明了在本章中讨论的以下两个日期和时间概念：

- 获取当前日期和时间并提取小时、分钟和秒的值
- 使用 Timer 设置应用程序的运行速度

要获取该范例的应用程序文件，请访问
www.adobe.com/go/learn_programmingAS3samples_flash_cn。
可以在 Samples/SimpleClock 文件夹中找到 SimpleClock 应用程序文件。该应用程序包含以下文件：

| 文件 | 描述 |
| --- | --- |
| SimpleClockApp.mxml<br>或<br>SimpleClockApp.fla | Flash 或 Flex 中的主应用程序文件 （分别为 FLA 和 MXML）。 |
| com/example/programmingas3/simpleclock/SimpleClock.as | 主应用程序文件。 |
| com/example/programmingas3/simpleclock/AnalogClockFace.as | 根据时间绘制一个圆形的钟面以及时针、分针和秒针。 |

## 定义 SimpleClock 类

此时钟示例很简单，但是将即使很简单的应用程序也组织得十分有条理是一种很好的做法，以便您将来能够很轻松地扩展这些应用程序。为此， SimpleClock 应用程序使用 SimpleClock 类处理启动和时间保持任务，然后使用另一个名称为 AnalogClockFace 的类来实际显示该时间。

以下代码用于定义和初始化 SimpleClock 类（请注意，在 Flash 版本中，SimpleClock 扩展了 Sprite 类）：

```
public class SimpleClock extends UIComponent
{
    /**
     * 时间显示组件。
     */
    private var face:AnalogClockFace;

    /**
     * Timer 的作用就像是应用程序的心跳。
     */
    private var ticker:Timer;
```

该类具有两个重要的属性:

- `face` 属性，它是 AnalogClockFace 类的实例
- `ticker` 属性，它是 Timer 类的实例

SimpleClock 类使用默认构造函数。`initClock()` 方法处理实际的设置工作，它创建钟面并启动 Timer 实例的计时。

## 创建钟面

SimpleClock 代码中后面的几行代码创建用于显示时间的钟面:

```
    /**
     * 设置 SimpleClock 实例。
     */
    public function initClock(faceSize:Number = 200)
    {
        // 创建钟面并将其添加到显示列表中
        face = new AnalogClockFace(Math.max(20, faceSize));
        face.init();
        addChild(face);

        // 绘制初始时钟显示
        face.draw();
```

可以将钟面的大小传递给 `initClock()` 方法。如果未传递 `faceSize` 值，则使用 200 个像素的默认大小。

接着，应用程序将钟面初始化，然后使用从 DisplayObject 类继承的 addChild() 方法将该钟面添加到显示列表。然后，它调用 AnalogClockFace.draw() 方法显示一次钟面，同时显示当前时间。

## 启动计时器

创建钟面后，initClock() 方法会设置一个计时器：

```
// 创建用来每秒触发一次事件的 Timer
ticker = new Timer(1000);

// 指定 onTick() 方法来处理 Timer 事件
ticker.addEventListener(TimerEvent.TIMER, onTick);

// 启动时钟计时
ticker.start();
```

首先，该方法初始化一个每秒（每隔 1000 毫秒）调度一次事件的 Timer 实例。由于没有向 Timer() 构造函数传递第二个 repeatCount 参数，因此 Timer 将无限期地重复计时。

SimpleClock.onTick() 方法将在每秒收到 timer 事件时执行一次。

```
public function onTick(event:TimerEvent):void
{
    // 更新时钟显示
    face.draw();
}
```

AnalogClockFace.draw() 方法仅绘制钟面和指针。

## 显示当前时间

AnalogClockFace 类中大多数代码都与设置钟面的显示元素有关。在对 AnalogClockFace 进行初始化时，它会绘制一个圆形轮廓，将数字文本标签放在每个小时刻度处，然后创建三个 Shape 对象，分别表示时钟的时针、分针和秒针。

在 SimpleClock 应用程序运行后，它会每秒调用一次 AnalogClockFace.draw() 方法，如下所示：

```
/**
 * 在绘制时钟显示时由父容器进行调用。
 */
public override function draw():void
{
    // 将当前日期和时间存储在实例变量中
    currentTime = new Date();
    showTime(currentTime);
}
```

此方法将当前时间保存在变量中，因此在绘制时钟指针的过程中无法改变时间。然后，调用 `showTime()` 方法以显示指针，如下所示：

```
/**
 * 以看起来不错的老式模拟时钟样式显示指定的 Date/Time。
 */
public function showTime(time:Date):void
{
    // 获取时间值
    var seconds:uint = time.getSeconds();
    var minutes:uint = time.getMinutes();
    var hours:uint = time.getHours();

    // 乘以 6 得到度数
    this.secondHand.rotation = 180 + (seconds * 6);
    this.minuteHand.rotation = 180 + (minutes * 6);

    // 乘以 30 得到基本度数，然后
    // 最多加上 29.5 度 (59 * 0.5)
    // 以计算分钟。
    this.hourHand.rotation = 180 + (hours * 30) + (minutes * 0.5);
}
```

首先，此方法提取当前时间的小时、分钟和秒的值。然后使用这些值来计算每个指针的角度。由于秒针会在 60 秒内旋转一圈，因此它每秒都会旋转 6 度 (360/60)。分针每分钟都旋转同样的度数。

时针每分钟都在更新，因此时针能够随着分针的跳动显示出某些时间变化过程。时针每小时旋转 30 度 (360/12)，但也会每分钟旋转半度（30 度除以 60 分钟）。

第 6 章

# 处理字符串

6

String 类包含使您能够使用文本字符串的方法。在处理许多对象时，字符串都十分重要。本章介绍的方法对于处理在对象（如 TextField、 StaticText、 XML、 ContextMenu 和 FileReference 对象）中使用的字符串很有帮助。

字符串是字符的序列。 ActionScript 3.0 支持 ASCII 字符和 Unicode 字符。

## 目录

# 字符串基础知识

## 处理字符串简介

在编程语言中，字符串是指一个文本值，即串在一起而组成单个值的一系列字母、数字或其它字符。例如，以下一行代码创建一个数据类型为 String 的变量，并为该变量赋予一个文本字符串值：

```
var albumName:String = "Three for the money";
```

正如此示例所示，在 ActionScript 中，可使用双引号或单引号将本文引起来以表示字符串值。以下是另外几个字符串示例：

```
"Hello"
"555-7649"
"http://www.adobe.com/"
```

每当在 ActionScript 中处理一段文本时，您都会用到字符串值。 ActionScript String 类是一种可用来处理文本值的数据类型。 String 实例通常用于很多其它 ActionScript 类中的属性、方法参数等。

## 常见的字符串处理任务

以下是本章中讨论的与字符串有关的常见任务：

- 创建 String 对象
- 处理特殊字符，如回车符、制表符和非键盘字符
- 测量字符串长度
- 隔离字符串中的各个字符
- 连接字符串
- 比较字符串
- 查找、提取以及替换字符串的各部分
- 使字符串变为大写或小写

## 重要概念和术语

以下参考列表包含您将会在本章中遇到的重要术语：

- ASCII：用于在计算机程序中表示文本字符和符号的系统。ASCII 系统支持 26 个字母英文字母表，以及有限的一组其它字符。
- 字符 (Character)：文本数据的最小单位（单个字母或符号）。
- 连接 (Concatenation)：通过将一个字符串值添加到另一个字符串值结尾，将多个字符串值连接在一起，从而创建一个新的字符串值。
- 空字符串 (Empty string)：不包含任何文本、空白或其它字符的字符串，可以写为 `""`。空字符串值不同于具有空值的 String 变量；空 String 变量是指没有赋予 String 实例的变量，而空字符串则包含一个实例，其值不包含任何字符。
- 字符串 (String)：一个文本值（字符序列）。
- 字符串文本或文本字符串（String literal 或 literal string）：在代码中显式编写的字符串值，编写为用双引号或单引号引起来的文本值。
- 子字符串 (Substring)：作为另一个字符串一部分的字符串。
- Unicode：用于在计算机程序中表示文本字符和符号的标准系统。Unicode 系统允许使用任何编写系统中的任何字符。

## 完成本章中的示例

学习本章的过程中，您可能想要自己动手测试一些示例代码清单。由于本章中的代码清单主要用于操作文本，测试示例将涉及查看示例中使用的变量值，方法是：通过将值写入舞台上的文本字段实例，或使用 `trace()` 函数将值输出到“输出”面板。第 53 页的“测试本章内的示例代码清单”中对这些技术进行了详细说明。

# 创建字符串

在 ActionScript 3.0 中，String 类用于表示字符串（文本）数据。ActionScript 字符串既支持 ASCII 字符也支持 Unicode 字符。创建字符串的最简单方式是使用字符串文本。要声明字符串文本，请使用双直引号 (") 或单直引号 (') 字符。例如，以下两个字符串是等效的:

```
var str1:String = "hello";
var str2:String = 'hello';
```

您还可以使用 `new` 运算符来声明字符串，如下所示：

```
var str1:String = new String("hello");
var str2:String = new String(str1);
var str3:String = new String();        // str3 == ""
```

下面的两个字符串是等效的:

```
var str1:String = "hello";
var str2:String = new String("hello");
```

要在使用单引号 (') 分隔符定义的字符串文本内使用单引号 ('),请使用反斜杠转义符 (\)。类似地,要在使用双引号 (") 分隔符定义的字符串文本内使用双引号 ("),请使用反斜杠转义符 (\)。下面的两个字符串是等效的:

```
var str1:String = "That's \"A-OK\"";
var str2:String = 'That\'s "A-OK"';
```

您可以根据字符串文本中存在的任何单引号或双引号来选择使用单引号或双引号,如下所示:

```
var str1:String = "ActionScript <span class='heavy'>3.0</span>";
var str2:String = '<item id="155">banana</item>';
```

请务必记住 ActionScript 可区分单直引号 (') 和左右单引号 (‘ 或 ’)。对于双引号也同样如此。请使用直引号来分割字符串文本。在将文本从其它来源粘贴到 ActionScript 中时,请确保使用正确的字符。

如下表所示,可以使用反斜杠转义符 (\) 在字符串文本中定义其它字符:

| 转义序列 | 字符 |
| --- | --- |
| `\b` | 退格符 |
| `\f` | 换页符 |
| `\n` | 换行符 |
| `\r` | 回车符 |
| `\t` | 制表符 |
| `\u`*nnnn* | Unicode 字符,字符代码由十六进制数字 *nnnn* 指定;例如,`\u263a` 为笑脸字符。 |
| `\x`*nn* | ASCII 字符,字符代码由十六进制数字 *nn* 指定。 |
| `\'` | 单引号 |
| `\"` | 双引号 |
| `\\` | 单个反斜杠字符 |

# length 属性

每个字符串都有 `length` 属性，其值等于字符串中的字符数：

```
var str:String = "Adobe";
trace(str.length);            // 输出：5
```

空字符串和 null 字符串的长度均为 0，如下例所示：

```
var str1:String = new String();
trace(str1.length);           // 输出：0

str2:String = '';
trace(str2.length);           // 输出：0
```

# 处理字符串中的字符

字符串中的每个字符在字符串中都有一个索引位置（整数）。第一个字符的索引位置为 0。例如，在以下字符串中，字符 `y` 的位置为 0，而字符 `w` 的位置为 5：

```
"yellow"
```

您可以使用 `charAt()` 方法和 `charCodeAt()` 方法检查字符串各个位置上的字符：

```
var str:String = "hello world!";
for (var:i = 0; i < str.length; i++)
{
  trace(str.charAt(i), "-", str.charCodeAt(i));
}
```

在运行此代码时，会产生如下输出：

```
h - 104
e - 101
l - 108
l - 108
o - 111
  - 32
w - 119
o - 111
r - 114
l - 108
d - 100
! - 33
```

此外，您还可以通过字符代码，使用 `fromCharCode()` 方法定义字符串，如下例所示：

```
var myStr:String =
  String.fromCharCode(104,101,108,108,111,32,119,111,114,108,100,33);
    // 将 myStr 设置为“hello world!”
```

# 比较字符串

可以使用以下运算符比较字符串：<、<=、!=、==、=> 和 >。可以将这些运算符与条件语句（如 if 和 while）一起使用，如下例所示：

```
var str1:String = "Apple";
var str2:String = "apple";
if (str1 < str2)
{
  trace("A < a, B < b, C < c, ...");
}
```

在将这些运算符用于字符串时，ActionScript 会使用字符串中每个字符的字符代码值从左到右比较各个字符，如下所示：

```
trace("A" < "B"); // true
trace("A" < "a"); // true
trace("Ab" < "az"); // true
trace("abc" < "abza"); // true
```

使用 == 和 != 运算符可比较两个字符串，也可以将字符串与其它类型的对象进行比较，如下例所示：

```
var str1:String = "1";
var str1b:String = "1";
var str2:String = "2";
trace(str1 == str1b); // true
trace(str1 == str2); // false
var total:uint = 1;
trace(str1 == total); // true
```

# 获取其它对象的字符串表示形式

可以获取任何类型对象的字符串表示形式。所有对象都提供了 toString() 方法来实现此目的：

```
var n:Number = 99.47;
var str:String = n.toString();
  // str == "99.47"
```

在使用 + 连接运算符连接 String 对象和不属于字符串的对象时，无需使用 toString() 方法。有关连接的详细信息，请参阅下一节。

对于给定对象，String() 全局函数返回的值与调用该对象的 toString() 方法返回的值相同。

# 连接字符串

字符串连接的含义是：将两个字符串按顺序合并为一个字符串。例如，可以使用 + 运算符来连接两个字符串：

```
var str1:String = "green";
var str2:String = "ish";
var str3:String = str1 + str2; // str3 == "greenish"
```

还可以使用 += 运算符来得到相同的结果，如下例所示：

```
var str:String = "green";
str += "ish"; // str == "greenish"
```

此外， String 类还包括 concat() 方法，可按如下方式对其进行使用：

```
var str1:String = "Bonjour";
var str2:String = "from";
var str3:String = "Paris";
var str4:String = str1.concat(" ", str2, " ", str3);
// str4 == "Bonjour from Paris"
```

如果使用 + 运算符（或 += 运算符）对 String 对象和 "非" 字符串的对象进行运算，ActionScript 会自动将非字符串对象转换为 String 对象以计算该表达式，如下例所示：

```
var str:String = "Area = ";
var area:Number = Math.PI * Math.pow(3, 2);
str = str + area; // str == "Area = 28.274333882308138"
```

但是，可以使用括号进行分组，为 + 运算符提供运算的上下文，如下例所示：

```
trace("Total: $" + 4.55 + 1.45); // 输出: Total: $4.551.45
trace("Total: $" + (4.55 + 1.45)); // 输出: Total: $6
```

# 在字符串中查找子字符串和模式

子字符串是字符串内的字符序列。例如，字符串 "abc" 具有如下子字符串："" 、"a"、"ab"、"abc"、 "b"、 "bc"、 "c"。可使用 ActionScript 的方法来查找字符串的子字符串。

模式是在 ActionScript 中通过字符串或正则表达式定义的。例如，下面的正则表达式定义了一个特定模式：字母 A、B 和 C 的后面跟着一个数字字符（正斜杠是正则表达式的分隔符）：

```
/ABC\d/
```

ActionScript 提供了在字符串中查找模式的方法，以及使用替换子字符串替换找到的匹配项的方法。随后的章节将介绍这些方法。

正则表达式可定义十分复杂的模式。有关详细信息，请参阅第 243 页的第 9 章 "使用正则表达式"。

## 通过字符位置查找子字符串

substr() 和 substring() 方法非常类似。两个方法都返回字符串的一个子字符串。并且两个方法都具有两个参数。在这两个方法中，第一个参数是给定字符串中起始字符的位置。不过，在 substr() 方法中，第二个参数是要返回的子字符串的 "长度"，而在 substring() 方法中，第二个参数是子字符串的 "结尾"处字符的位置 （该字符未包含在返回的字符串中）。此示例显示了这两种方法之间的差别：

```
var str:String = "Hello from Paris, Texas!!!";
trace(str.substr(11,15)); // 输出: Paris, Texas!!!
trace(str.substring(11,15)); // 输出: Pari
```

slice() 方法的功能类似于 substring() 方法。当指定两个非负整数作为参数时，其运行方式将完全一样。但是，slice() 方法可以使用负整数作为参数，此时字符位置将从字符串末尾开始向前算起，如下例所示：

```
var str:String = "Hello from Paris, Texas!!!";
trace(str.slice(11,15)); // 输出: Pari
trace(str.slice(-3,-1)); // 输出: !!
trace(str.slice(-3,26)); // 输出: !!!
trace(str.slice(-3,str.length)); // 输出: !!!
trace(str.slice(-8,-3)); // 输出: Texas
```

可以结合使用非负整数和负整数作为 slice() 方法的参数。

## 查找匹配子字符串的字符位置

可以使用 indexOf() 和 lastIndexOf() 方法在字符串内查找匹配的子字符串，如下例所示：

```
var str:String = "The moon, the stars, the sea, the land";
trace(str.indexOf("the")); // 输出: 10
```

请注意， indexOf() 方法区分大小写。

可以指定第二个参数以指出在字符串中开始进行搜索的起始索引位置，如下所示：

```
var str:String = "The moon, the stars, the sea, the land"
trace(str.indexOf("the", 11)); // 输出: 21
```

lastIndexOf() 方法在字符串中查找子字符串的最后一个匹配项：

```
var str:String = "The moon, the stars, the sea, the land"
trace(str.lastIndexOf("the")); // 输出: 30
```

如果为 lastIndexOf() 方法提供了第二个参数，搜索将从字符串中的该索引位置反向 （从右到左）进行：

```
var str:String = "The moon, the stars, the sea, the land"
trace(str.lastIndexOf("the", 29)); // 输出: 21
```

## 创建由分隔符分隔的子字符串数组

可使用 split() 方法创建子字符串数组，该数组根据分隔符进行划分。例如，可以将逗号分隔或制表符分隔的字符串分为多个字符串。

以下示例说明如何使用 "与" 字符 (&) 作为分隔符，将数组分割为多个子字符串：

```
var queryStr:String = "first=joe&last=cheng&title=manager&StartDate=3/6/65";
var params:Array = queryStr.split("&", 2); // params ==
  ["first=joe","last=cheng"]
```

split() 方法的第二个参数是可选参数，该参数定义所返回数组的最大大小。

此外，还可以使用正则表达式作为分隔符：

```
var str:String = "Give me\t5."
var a:Array = str.split(/\s+/); // a == ["Give","me","5."]
```

有关详细信息，请参阅第 243 页的第 9 章 "使用正则表达式"和《ActionScript 3.0 语言和组件参考》。

## 在字符串中查找模式并替换子字符串

String 类提供了在字符串中处理模式的以下方法：

- 使用 match() 和 search() 方法可查找与模式相匹配的子字符串。
- 使用 replace() 方法可查找与模式相匹配的子字符串并使用指定子字符串替换它们。

随后的章节将介绍这些方法。

您可以使用字符串或正则表达式定义在这些方法中使用的模式。有关正则表达式的详细信息，请参阅第 243 页的第 9 章 "使用正则表达式"。

### 查找匹配的子字符串

search() 方法返回与给定模式相匹配的第一个子字符串的索引位置，如下例所示：

```
var str:String = "The more the merrier.";
// （此搜索区分大小写。）
trace(str.search("the")); // 输出: 9
```

您还可以使用正则表达式定义要匹配的模式，如下例所示：

```
var pattern:RegExp = /the/i;
var str:String = "The more the merrier.";
trace(str.search(pattern)); // 0
```

trace() 方法的输出为 0，因为字符串中第一个字符的索引位置为 0。在正则表达式中设置了 i 标志，因此搜索时不区分大小写。

search() 方法仅查找一个匹配项并返回其起始索引位置，即便在正则表达式中设置了 g（全局）标志。

下例展示一个更复杂的正则表达式，该表达式匹配被双引号引起来的字符串：

```
var pattern:RegExp = /"[^"]*"/;
var str:String = "The \"more\" the merrier.";
trace(str.search(pattern)); // 输出: 4

str = "The \"more the merrier.";
trace(str.search(pattern)); // 输出: -1
// （指示没有任何匹配项，因为不存在任何结束双引号。）
```

`match()` 方法的工作方式与此类似。它搜索一个匹配的子字符串。但是，如果在正则表达式模式中使用了全局标志 （如下例所示）， `match()` 将返回一个包含匹配子字符串的数组：

```
var str:String = "bob@example.com, omar@example.org";
var pattern:RegExp = /\w*@\w*\.[org|com]+/g;
var results:Array = str.match(pattern);
```

`results` 数组被设置为以下内容：

```
["bob@example.com","omar@example.org"]
```

有关正则表达式的详细信息，请参阅第 243 页的第 9 章 “使用正则表达式”。

## 替换匹配的子字符串

您可以使用 `replace()` 方法在字符串中搜索指定模式并使用指定的替换字符串替换匹配项，如下例所示：

```
var str:String = "She sells seashells by the seashore.";
var pattern:RegExp = /sh/gi;
trace(str.replace(pattern, "sch"));
  //sche sells seaschells by the seaschore.
```

请注意，在本例中，因为在正则表达式中设置了 `i` (`ignoreCase`) 标志，所以匹配的字符串是不区分大小写的；而且因为设置了 `g` (`global`) 标志，所以会替换多个匹配项。有关详细信息，请参阅第 243 页的第 9 章 “使用正则表达式”。

可以在替换字符串中包括以下 `$` 替换代码。下表中显示的替换文本将被插入并替换 `$` 替换代码：

| $ 代码 | 替换文本 |
|---|---|
| `$$` | $ |
| `$&` | 匹配的子字符串。 |
| `` $` `` | 字符串中位于匹配的子字符串前面的部分。该代码使用左单直引号字符 (`) 而不是单直引号 (') 或左单弯引号 (‘)。 |
| `$'` | 字符串中位于匹配的子字符串后的部分。该代码使用单直引号 (’)。 |

| $ 代码 | 替换文本 |
|---|---|
| `$n` | 第 *n* 个捕获的括号组匹配项，其中 *n* 是 1-9 之间的数字，并且 `$n` 后面没有十进制数字。 |
| `$nn` | 第 *nn* 个捕获的括号组匹配项，其中 *nn* 是一个十进制的两位数 (01-99)。如果未定义第 *nn* 个捕获内容，则替换文本为空字符串。 |

例如，下面说明了如何使用 `$2` 和 `$1` 替换代码，它们分别表示第一个和第二个匹配的捕获组：

```
var str:String = "flip-flop";
var pattern:RegExp = /(\w+)-(\w+)/g;
trace(str.replace(pattern, "$2-$1")); // flop-flip
```

也可以使用函数作为 `replace()` 方法的第二个参数。匹配的文本将被函数的返回值替换。

```
var str:String = "Now only $9.95!";
var price:RegExp = /\$([\d,]+.\d+)+/i;
trace(str.replace(price, usdToEuro));

function usdToEuro(matchedSubstring:String,
                   capturedMatch1:String,
                   index:int,
                   str:String):String
{
  var usd:String = capturedMatch1;
  usd = usd.replace(",", "");
  var exchangeRate:Number = 0.853690;
  var euro:Number = usd * exchangeRate;
  const euroSymbol:String = String.fromCharCode(8364);
  return euro.toFixed(2) + " " + euroSymbol;
}
```

在使用函数作为 `replace()` 方法的第二个参数时，将向该函数传递如下参数：

- 字符串的匹配部分。
- 任何捕获的括号组匹配项。按这种方式传递的参数数目因括号匹配项的数目而异。您可以在函数代码中通过检查 `arguments.length - 3` 来确定括号匹配项的数目。
- 字符串中匹配开始的索引位置。
- 完整的字符串。

# 在大小写之间转换字符串

如下例所示，toLowerCase() 方法和 toUpperCase() 方法分别将字符串中的英文字母字符转换为小写和大写：

```
var str:String = "Dr. Bob Roberts, #9."
trace(str.toLowerCase()); // dr. bob roberts, #9.
trace(str.toUpperCase()); // DR. BOB ROBERTS, #9.
```

执行完这些方法后，源字符串仍保持不变。要转换源字符串，请使用下列代码：

```
str = str.toUpperCase();
```

这些方法可处理扩展字符，而并不仅限于 a-z 和 A-Z：

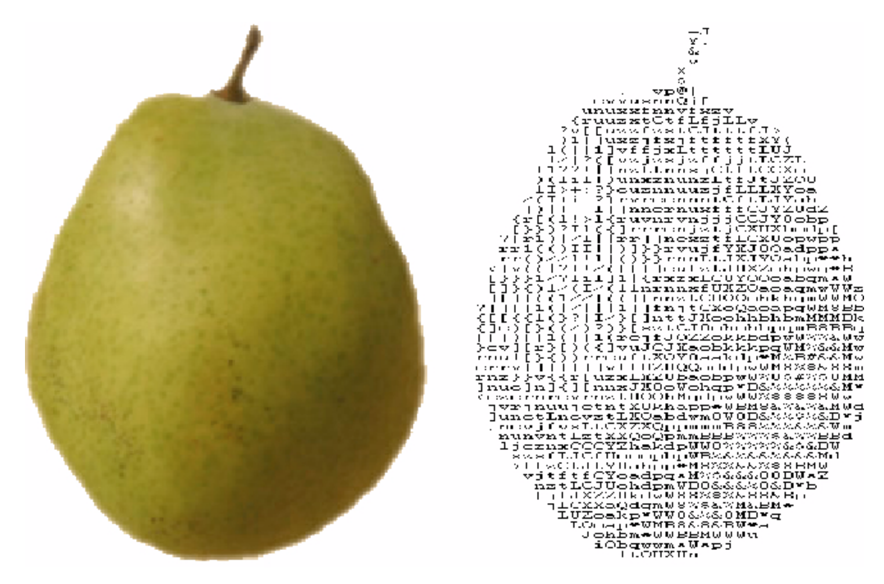
```
var str:String = "José Barça";
trace(str.toUpperCase(), str.toLowerCase()); // JOSÉ BARÇA josé barça
```

# 示例：ASCII 字符图

此 ASCII 字符图示例说明了可以在 ActionScript 3.0 中使用 String 类实现的大量功能，其中包括：

- 使用 String 类的 split() 方法可从由某个字符分隔的字符串中提取值（制表符分隔的文本文件中的图像信息）。
- 使用多种字符串操作技术（包括 split()、连接，以及使用 substring() 和 substr() 提取字符串的一部分）可将图像标题中每个单词的第一个字母变为大写形式。
- 使用 getCharAt() 方法可从字符串中获取单个字符（以确定对应于某个灰度位图值的 ASCII 字符）。
- 使用字符串连接可以按一次一个字符的方式建立图像的 ASCII 字符图表示形式。

“ASCII 字符图”这一术语指的是图像的文本表示形式，即使用等宽字体字符（如 Courier New 字符）的网格来绘制图像。下图便是该应用程序所生成 ASCII 字符图的一个例子：



*图形的 ASCII 字符图版本显示在右侧。*

要获取该范例的应用程序文件，请访问 www.adobe.com/go/learn_programmingAS3samples_flash_cn。可以在文件夹 Samples/AsciiArt 下找到 ASCIIArt 应用程序文件。该应用程序包含以下文件：

| 文件 | 说明 |
|---|---|
| AsciiArtApp.mxml<br>或<br>AsciiArtApp.fla | Flash (FLA) 或 Flex (MXML) 中的主应用程序文件。 |
| com/example/programmingas3/asciiArt/AsciiArtBuilder.as | 此类提供了应用程序主要功能，包括了从文本文件中提取图像元数据、加载图像和管理图像到文本的转换过程等功能。 |
| com/example/programmingas3/asciiArt/BitmapToAsciiConverter.as | 此类提供了用于将图像数据转换为字符串版本的 `parseBitmapData()` 方法。 |
| com/example/programmingas3/asciiArt/Image.as | 此类表示所加载的位图图像。 |
| com/example/programmingas3/asciiArt/ImageInfo.as | 此类表示 ASCII 字符图像的元数据（如标题、图像文件 URL 等）。 |

| 文件 | 说明 |
| --- | --- |
| image/ | 此文件夹包含应用程序使用的图像。 |
| txt/ImageData.txt | 此制表符分隔的文本文件中包含与应用程序要加载的图像有关的信息。 |

## 提取由制表符分隔的值

此示例使用了将应用程序数据与应用程序本身分开存储的通行做法；通过这种方式，在数据发生更改时（例如，添加了另一幅图像或更改了图像标题），无需重新创建该 SWF 文件。在本例中，图像元数据（包括图像标题、实际图像文件的 URL 以及用来操作图像的某些值）存储在一个文本文件中（项目中的 txt/ImageData.txt 文件）。该文本文件的内容如下所示：

```
FILENAMETITLEWHITE_THRESHHOLDBLACK_THRESHHOLD
FruitBasket.jpgPear, apple, orange, and bananad810
Banana.jpgA picture of a bananaC820
Orange.jpgorangeFF20
Apple.jpgpicture of an apple6E10
```

文件使用特定的制表符分隔格式。第一行为标题行。其余行包含要加载的每个位图的如下数据：

- 位图的文件名。
- 位图的显示名称。
- 位图的白色阀值和黑色阀值。这些值是十六进制值，高于这些值或低于这些值的像素将分别被认为是全白或全黑的。

应用程序启动后，AsciiArtBuilder 类将加载并分析文本文件的内容，以便创建它要显示的图像的 “堆栈”，执行这些操作时使用 AsciiArtBuilder 类的 `parseImageInfo()` 方法中的如下代码：

```
var lines:Array = _imageInfoLoader.data.split("\n");
var numLines:uint = lines.length;
for (var i:uint = 1; i < numLines; i++)
{
    var imageInfoRaw:String = lines[i];
    ...
    if (imageInfoRaw.length > 0)
    {
        // 创建一条新的图像信息记录并将其添加到图像信息数组中。
        var imageInfo:ImageInfo = new ImageInfo();

        // 将当前行分割为由制表符（\t）分隔的多个值
        // 并且提取出各个属性：
        var imageProperties:Array = imageInfoRaw.split("\t");
        imageInfo.fileName = imageProperties[0];
```

```
        imageInfo.title = normalizeTitle(imageProperties[1]);
        imageInfo.whiteThreshold = parseInt(imageProperties[2], 16);
        imageInfo.blackThreshold = parseInt(imageProperties[3], 16);
        result.push(imageInfo);
    }
}
```

文本文件的完整内容包含在单个 String 实例中，即 _imageInfoLoader.data 属性。使用 split() 方法并以换行符 (\n) 为参数，将该 String 实例分割到一个 Array (lines) 中，数组元素为文本文件的各个行。然后，代码使用一个循环处理每行（第一行除外，因为它只包含标题而不包含实际内容）。在循环内部，再一次使用 split() 方法将每行的内容分割到一组值（名为 imageProperties 的 Array 对象）中。在本例中，用于 split() 方法的参数为制表符 (\t)，因为每行中的值均由制表符进行分隔。

## 使用 String 的方法标准化图像标题

此应用程序的设计目标之一便是使用一种标准格式显示所有图像标题，即标题中每一个单词的第一个字母均为大写形式（在英文标题中通常不大写的少数单词除外）。应用程序并不假定文本文件已经包含格式正确的标题，它在从文本文件中提取标题时对标题格式进行设置。

在前面的代码清单中，使用了如下代码行来提取每个图像元数据值：

```
imageInfo.title = normalizeTitle(imageProperties[1]);
```

在该代码中，来自文本文件的图像标题在存储到 ImageInfo 对象之前先传递给 normalizeTitle() 方法进行处理：

```
private function normalizeTitle(title:String):String
{
    var words:Array = title.split(" ");
    var len:uint = words.length;
    for (var i:uint; i < len; i++)
    {
        words[i] = capitalizeFirstLetter(words[i]);
    }

    return words.join(" ");
}
```

该方法使用 split() 方法将标题分割为各个单独的单词（由空格字符加以分隔），然后将每个单词传递给 capitalizeFirstLetter() 方法进行处理，接着使用 Array 类的 join() 方法将单词重新合并为一个字符串。

如同其名称所表达的含义，capitalizeFirstLetter() 方法实际执行将每个单词的第一个字母变为大写形式的工作：

```
/**
 * 将单个单词的第一个字母变为大写形式，除非该单词属于
 * 英语中通常不大写的一组单词之一。
 */
private function capitalizeFirstLetter(word:String):String
{
  switch (word)
  {
    case "and":
    case "the":
    case "in":
    case "an":
    case "or":
    case "at":
    case "of":
    case "a":
      // 不对这些单词执行任何操作。
      break;
    default:
      // 对于其它任何单词，则会将其第一个字符变为大写形式。
      var firstLetter:String = word.substr(0, 1);
      firstLetter = firstLetter.toUpperCase();
      var otherLetters:String = word.substring(1);
      word = firstLetter + otherLetters;
  }
  return word;
}
```

在英语中，如果标题中某个单词为以下单词之一，则 "不会"将其首字符变为大写形式：and、the、in、an、or、at、of 或 a。(这是相关规则的简化版本。)为了实现此逻辑，代码首先使用 switch 语句来检查单词是否为不应将其首字符大写的单词之一。如果是，代码直接跳出 switch 语句。另一方面，如果单词应大写，则在几个步骤中完成此操作，如下所示：

1. 使用 substr(0, 1) 提取出单词的第一个字母，该命令从位于索引位置 0 的字符 (即字符串的第一个字母，由第一个参数 0 指定）开始提取子字符串。该子字符串的长度为一个字符 (由第二个参数 1 指定)。

2. 使用 toUpperCase() 方法将该字符变为大写形式。

3. 使用 substring(1) 提取原始单词的其余字符，该命令提取从索引位置 1（第二个字母）开始直至字符串结尾（通过将 substring() 方法的第二个参数保留为空进行指定）的子字符串。

4. 使用字符串连接 firstLetter + otherLetters 将刚才变为大写形式的第一个字母与其余字母合并在一起，创建出最终的单词。

## 生成 ASCII 字符图文本

BitmapToAsciiConverter 类提供了将位图图像转换为其 ASCII 文本表示形式的功能。此过程由 `parseBitmapData()` 方法执行，下面展示了该方法的部分工作过程：

```
var result:String = "";

// 自上向下遍历所有像素行：
for (var y:uint = 0; y < _data.height; y += verticalResolution)
{
    // 在每一行中，自左向右遍历所有像素：
    for (var x:uint = 0; x < _data.width; x += horizontalResolution)
    {
        ...

        // 将位于 0-255 范围内的灰度值转换为
        // 介于 0-64 之间的一个值 （因为这是可用字符集合中的
        // “灰度梯度”数目）：
        index = Math.floor(grayVal / 4);
        result += palette.charAt(index);
    }
    result += "\n";
}
return result;
```

此代码首先定义一个名为 `result` 的 String 实例，将使用它来构建位图图像的 ASCII 字符图版本。然后，它遍历源位图图像的每个像素。通过使用若干颜色处理技术 （为了简便起见，此处省略了对这些技术的介绍），它将每个像素的红色、绿色和蓝色值转换为单个灰度值 （一个介于 0-255 之间的数字）。接着，代码将该值除以 4 （如下所示）以将其转换为介于 0-63 之间的一个值，此值存储在变量 `index` 中。（之所以使用 0-63 的范围是因为此应用程序使用的可用 ASCII 字符的 “调色板”包含 64 个值。）该字符调色板在 BitmapToAsciiConverter 类中定义为一个 String 实例：

```
// 字符按从最暗到最亮的顺序排列，所以它们
// 在字符串中的位置 （索引）对应于一个相对颜色值
// （0 = 黑色）。
private static const palette:String =
    "@#$%&8BMW*mwqpdbkhaoQ0OZXYUJCLtfjzxnuvcr[]{}1()|/?Il!i><+_~-;,. ";
```

由于变量 `index` 定义调色板中的哪个 ASCII 字符对应于位图图像中的当前像素，可使用 `charAt()` 方法从 `palette` 字符串中检索该字符。然后，使用连接赋值运算符 (+=) 将其追加到 `result` 字符串实例。此外，在每行像素的末尾，会将一个换行符连接到 `result` 字符串的末尾，强制该行换行以创建新的一行字符 “像素”。

第 7 章

# 处理数组

7

使用数组可以在单数据结构中存储多个值。可以使用简单的索引数组（使用固定有序整数索引存储值），也可以使用复杂的关联数组 （使用任意键存储值）。数组也可以是多维的，即包含本身是数组的元素。本章讨论如何创建和操作各种类型的数组。

## 目录



# 数组基础知识

## 处理数组简介

通常，您需要在编程中使用一组项目，而不是单个对象；例如，在音乐播放器应用程序中，您可能希望将等待播放的歌曲放在列表中。您不希望必须为该列表中的每首歌曲创建单独的变量，而是希望将所有 Song 对象放在一个包中，因而能够将其作为一个组进行使用。

数组是一种编程元素，它用作一组项目的容器，如一组歌曲。通常，数组中的所有项目都是相同类的实例，但这在 ActionScript 中并不是必需的。数组中的各个项目称为数组的 “元素”。可以将数组视为变量的 “文件柜”。您可以将变量作为元素添加到数组中，就像将文件夹放到文件柜中一样。当文件柜中包含一些文件后，您可以将数组作为单个变量使用（就像将整个文件柜搬到其它地方一样）；将这些变量作为组使用（就像逐个浏览文件夹以搜索一条信息一样）；或者，您可以分别访问它们 （就像打开文件柜并选择单个文件夹一样）。

例如，假设您要创建一个音乐播放器应用程序，用户可以在其中选择多首歌曲，并将这些歌曲添加到播放列表中。您可以在 ActionScript 代码中添加一个名为 `addSongsToPlaylist()` 的方法，该方法接受单个数组作为参数。无论要将多少首歌曲（几首、很多首甚至只有一首）添加到列表中，您都只需要调用一次 `addSongsToPlaylist()` 方法，并将其传递给包含 Song 对象的数组。在 `addSongsToPlaylist()` 方法中，可以使用循环来逐个访问数组元素（歌曲），并将歌曲实际添加到播放列表中。

最常见的 ActionScript 数组类型是“索引数组”，此数组将每个项目存储在编号位置（称为“索引”），您可以使用该编号来访问项目，如地址。Array 类用于表示索引数组。索引数组可以很好地满足大多数编程需要。索引数组的一个特殊用途是多维数组，此索引数组的元素也是索引数组（这些数组又包含其它元素）。另一种数组类型是“关联数组”，该数组使用字符串“键”来标识各个元素，而不是使用数字索引。最后，对于高级用户，ActionScript 3.0 还包括 Dictionary 类（表示“字典”），在此数组中，您可以将任何类型的对象用作键来区分元素。

## 常见数组任务

本章介绍了以下常见的数组使用活动：

- 创建索引数组
- 添加和删除数组元素
- 对数组元素进行排序
- 提取数组的某些部分
- 处理关联数组和字典
- 处理多维数组
- 复制数组元素
- 创建数组子类

## 重要概念和术语

以下参考列表包含将会在本章中遇到的重要术语：

- 数组 (Array)：用作容器以将多个对象组合在一起的对象。
- 关联数组 (Associative array)：使用字符串键来标识各个元素的数组。
- 字典 (Dictionary)：其项目由一对对象（称为键和值）组成的数组。它使用键来标识单个元素，而不是使用数字索引。
- 元素 (Element)：数组中的单个项目。
- 索引 (Index)：用于标识索引数组中的单个元素的数字“地址”。

■ 索引数组 (Indexed array): 这是一种标准类型的数组，它将每个元素存储在编号元素中，并使用数字（索引）来标识各个元素。
■ 键 (Key)：用于标识关联数组或字典中的单个元素的字符串或对象。
■ 多维数组 (Multidimensional array)：此数组包含的项目是数组，而不是单个值。

## 完成本章中的示例

学习本章的过程中，您可能想要自己动手测试一些示例代码清单。实质上本章中的代码清单包括适当的 `trace()` 函数调用。要测试本章中的代码清单，请执行以下操作：

1. 创建一个空的 Flash 文档。
2. 在时间轴上选择一个关键帧。
3. 打开 “动作” 面板，将代码清单复制到 “脚本” 窗格中。
4. 使用 “控制” > “测试影片” 运行程序。

   可在 “输出” 面板中看到 `trace()` 函数的结果。

第 53 页的 “测试本章内的示例代码清单”中对用于测试示例代码清单的此技术和其它技术进行了详细说明。

# 索引数组

索引数组存储一系列经过组织的单个或多个值，其中的每个值都可以通过使用一个无符号整数值进行访问。第一个索引始终是数字 0，且添加到数组中的每个后续元素的索引以 1 为增量递增。正如以下代码所示，可以调用 Array 类构造函数或使用数组文本初始化数组来创建索引数组：

```
// 使用 Array 构造函数。
var myArray:Array = new Array();
myArray.push("one");
myArray.push("two");
myArray.push("three");
trace(myArray); // 输出: one,two,three

// 使用数组文本。
var myArray:Array = ["one", "two", "three"];
trace(myArray); // 输出: one,two,three
```

Array 类中还包含可用来修改索引数组的属性和方法。这些属性和方法几乎是专用于索引数组而非关联数组的。

索引数组使用无符号 32 位整数作为索引号。索引数组的最大大小为 $2^{32}$-1，即 4,294,967,295。如果要创建的数组大小超过最大值，则会出现运行时错误。

数组元素的值可以为任意数据类型。ActionScript 3.0 不支持“指定类型的数组”概念，也就是说，不能指定数组的所有元素都属于特定数据类型。

本部分说明如何使用 Array 类创建和修改索引数组，首先讲的是如何创建数组。修改数组的方法分为三类，包括如何插入元素、删除元素和对数组进行排序。最后一类中的方法将现有数组当作只读数组，这些方法仅用于查询数组。所有查询方法都返回新的数组，而非修改现有数组。本部分结尾讨论了如何扩展 Array 类。

## 创建数组

Array 构造函数的使用有三种方式。第一种，如果调用不带参数的构造函数，会得到空数组。可以使用 Array 类的 `length` 属性来验证数组是否不包含元素。例如，以下代码调用不带参数的 Array 构造函数：

```
var names:Array = new Array();
trace(names.length); // 输出: 0
```

第二种，如果将一个数字用作 Array 构造函数的唯一参数，则会创建长度等于此数值的数组，并且每个元素的值都设置为 `undefined`。参数必须为介于值 0 和 4,294,967,295 之间的无符号整数。例如，以下代码调用带有一个数字参数的 Array 构造函数：

```
var names:Array = new Array(3);
trace(names.length); // 输出: 3
trace(names[0]); // 输出: undefined
trace(names[1]); // 输出: undefined
trace(names[2]); // 输出: undefined
```

第三种，如果调用构造函数并传递一个元素列表作为参数，将创建具有与每个参数对应的元素的数组。以下代码将三个参数传递给 Array 构造函数：

```
var names:Array = new Array("John", "Jane", "David");
trace(names.length); // 输出: 3
trace(names[0]); // 输出: John
trace(names[1]); // 输出: Jane
trace(names[2]); // 输出: David
```

也可以创建具有数组文本或对象文本的数组。可以将数组文本直接分配给数组变量，如以下示例所示：

```
var names:Array = ["John", "Jane", "David"];
```

## 插入数组元素

可以使用 Array 类的三种方法（push()、unshift() 和 splice()）将元素插入数组。push() 方法用于在数组末尾添加一个或多个元素。换言之，使用 push() 方法在数组中插入的最后一个元素将具有最大索引号。unshift() 方法用于在数组开头插入一个或多个元素，并且始终在索引号 0 处插入。splice() 方法用于在数组中的指定索引处插入任意数目的项目。

下面的示例对所有三种方法进行了说明。它创建一个名为 planets 的数组，以便按照距离太阳的远近顺序存储各个行星的名称。首先，调用 push() 方法以添加初始项 Mars。接着，调用 unshift() 方法在数组开头插入项 Mercury。最后，调用 splice() 方法在 Mercury 之后和 Mars 之前插入项 Venus 和 Earth。传递给 splice() 的第一个参数是整数 1，它用于指示从索引 1 处开始插入。传递给 splice() 的第二个参数是整数 0，它表示不应删除任何项。传递给 splice() 的第三和第四个参数 Venus 和 Earth 为要插入的项。

```
var planets:Array = new Array();
planets.push("Mars");       // 数组内容：Mars
planets.unshift("Mercury"); // 数组内容：Mercury,Mars
planets.splice(1, 0, "Venus", "Earth");
trace(planets);             // 数组内容：Mercury,Venus,Earth,Mars
```

push() 和 unshift() 方法均返回一个无符号整数，它们表示修改后的数组长度。在用于插入元素时，splice() 方法返回空数组，这看上去也许有点奇怪，但考虑到 splice() 方法的多用途性，您便会觉得它更有意义。通过使用 splice() 方法，不仅可以将元素插入到数组中，而且还可以从数组中删除元素。用于删除元素时，splice() 方法将返回包含被删除元素的数组。

## 删除数组元素

可以使用 Array 类的三种方法（pop()、shift() 和 splice()）从数组中删除元素。pop() 方法用于从数组末尾删除一个元素。换言之，它将删除位于最大索引号处的元素。shift() 方法用于从数组开头删除一个元素，也就是说，它始终删除索引号 0 处的元素。splice() 方法既可用来插入元素，也可以删除任意数目的元素，其操作的起始位置位于由发送到此方法的第一个参数指定的索引号处。

以下示例使用所有三种方法从数组中删除元素。它创建一个名为 oceans 的数组，以便存储较大水域的名称。数组中的某些名称为湖泊的名称而非海洋的名称，因此需要将其删除。

首先，使用 splice() 方法删除项 Aral 和 Superior，并插入项 Atlantic 和 Indian。传递给 splice() 的第一个参数是整数 2，它表示应从列表中的第三个项（即索引 2 处）开始执行操作。第二个参数 2 表示应删除两个项。其余两个参数 Atlantic 和 Indian 是要在索引 2 处插入的值。

然后，使用 pop() 方法删除数组中的最后一个元素 Huron。最后，使用 shift() 方法删除数组中的第一个项 Victoria。

```
var oceans:Array = ["Victoria", "Pacific", "Aral", "Superior", "Indian",
  "Huron"];
oceans.splice(2, 2, "Arctic", "Atlantic"); // 替换 Aral 和 Superior
oceans.pop();   // 删除 Huron
oceans.shift(); // 删除 Victoria
trace(oceans);  // 输出: Pacific,Arctic,Atlantic,Indian
```

pop() 和 shift() 方法均返回已删除的项。由于数组可以包含任意数据类型的值，因而返回值的数据类型为 Object。splice() 方法将返回包含被删除值的数组。可以更改 oceans 数组示例，以使 splice() 调用将此数组分配给新的数组变量，如以下示例所示：

```
var lakes:Array = oceans.splice(2, 2, "Arctic", "Atlantic");
trace(lakes); // 输出: Aral,Superior
```

您可能会遇到这样的代码，它在数组元素上使用 delete 运算符。delete 运算符用于将数组元素的值设置为 undefined，但它不会从数组中删除元素。例如，下面的代码在 oceans 数组的第三个元素上使用 delete 运算符，但此数组的长度仍然为 5：

```
var oceans:Array = ["Arctic", "Pacific", "Victoria", "Indian", "Atlantic"];
delete oceans[2];
trace(oceans);        // 输出: Arctic,Pacific,,Indian,Atlantic
trace(oceans[2]);     // 输出: undefined
trace(oceans.length); // 输出: 5
```

可以使用数组的 length 属性截断数组。如果将数组的 length 属性设置为小于数组当前长度的值，则会截断数组，在索引号高于 length 的新值减 1 处所存储的任何元素将被删除。例如，如果 oceans 数组的排序是将所有有效项放在数组的开始处，则可以使用 length 属性删除数组末尾的项，如以下代码所示：

```
var oceans:Array = ["Arctic", "Pacific", "Victoria", "Aral", "Superior"];
oceans.length = 2;
trace(oceans); // 输出: Arctic,Pacific
```

## 对数组排序

可以使用三种方法（reverse()、sort() 和 sortOn()）通过排序或反向排序来更改数组的顺序。所有这些方法都用来修改现有数组。reverse() 方法用于按照以下方式更改数组的顺序：最后一个元素变为第一个元素，倒数第二个元素变为第二个元素，依此类推。sort() 方法可用来按照多种预定义的方式对数组进行排序，甚至可用来创建自定义排序算法。sortOn() 方法可用来对对象的索引数组进行排序，这些对象具有一个或多个可用作排序键的公共属性。

reverse() 方法不带参数，也不返回值，但可以将数组从当前顺序切换为相反顺序。以下示例颠倒了 oceans 数组中列出的海洋顺序：

```
var oceans:Array = ["Arctic", "Atlantic", "Indian", "Pacific"];
oceans.reverse();
trace(oceans); // 输出：Pacific,Indian,Atlantic,Arctic
```

sort() 方法按照 “默认排序顺序” 重新安排数组中的元素。默认排序顺序具有以下特征：

- 排序区分大小写，也就是说大写字符优先于小写字符。例如，字母 D 优先于字母 b。
- 排序按照升序进行，也就是说低位字符代码（例如 A）优先于高位字符代码（例如 B）。
- 排序将相同的值互邻放置，并且不区分顺序。
- 排序基于字符串，也就是说，在比较元素之前，先将其转换为字符串（例如，10 优先于 3，因为相对于字符串 "3" 而言，字符串 "1" 具有低位字符代码）。

您也许需要不区分大小写或者按照降序对数组进行排序，或者您的数组中包含数字，从而需要按照数字顺序而非字母顺序进行排序。sort() 方法具有 options 参数，可通过该参数改变默认排序顺序的各个特征。options 是由 Array 类中的一组静态常量定义的，如以下列表所示：

- Array.CASEINSENSITIVE：此选项可使排序不区分大小写。例如，小写字母 b 优先于大写字母 D。
- Array.DESCENDING：用于颠倒默认的升序排序。例如，字母 B 优先于字母 A。
- Array.UNIQUESORT：如果发现两个相同的值，此选项将导致排序中止。
- Array.NUMERIC：这会导致排序按照数字顺序进行，比方说 3 优先于 10。

以下示例重点说明了这些选项中的某些选项。它创建一个名为 poets 的数组，并使用几种不同的选项对其进行排序。

```
var poets:Array = ["Blake", "cummings", "Angelou", "Dante"];
poets.sort(); // 默认排序
trace(poets); // 输出：Angelou,Blake,Dante,cummings

poets.sort(Array.CASEINSENSITIVE);
trace(poets); // 输出：Angelou,Blake,cummings,Dante

poets.sort(Array.DESCENDING);
trace(poets); // 输出：cummings,Dante,Blake,Angelou

poets.sort(Array.DESCENDING | Array.CASEINSENSITIVE); // 使用两个选项
trace(poets); // 输出：Dante,cummings,Blake,Angelou
```

您也可以编写自定义排序函数，然后将其作为参数传递给 sort() 方法。例如，如果有一个名称列表，其中每个列表元素都包含一个人的全名，但现在要按照姓来对列表排序，则必须使用自定义排序函数解析每个元素，然后使用排序函数中的姓。以下代码说明如何使用作为参数传递给 Array.sort() 方法的自定义函数来完成上述工作：

```
var names:Array = new Array("John Q. Smith", "Jane Doe", "Mike Jones");
function orderLastName(a, b):int
{
  var lastName:RegExp = /\b\S+$/;
  var name1 = a.match(lastName);
  var name2 = b.match(lastName);
  if (name1 < name2)
  {
    return -1;
  }
  else if (name1 > name2)
  {
    return 1;
  }
  else
  {
    return 0;
  }
}

trace(names); // 输出: John Q. Smith,Jane Doe,Mike Jones
names.sort(orderLastName);
trace(names); // 输出: Jane Doe,Mike Jones,John Q. Smith
```

自定义排序函数 orderLastName() 使用正则表达式从每个元素中提取姓，以用于比较操作。针对 names 数组调用 sort() 方法时，函数标识符 orderLastName 用作唯一的参数。排序函数接受两个参数 a 和 b，因为它每次对两个数组元素进行操作。排序函数的返回值指示应如何对元素排序：

- 返回值 -1 表示第一个参数 a 优先于第二个参数 b。
- 返回值 1 表示第二个参数 b 优先于第一个参数 a。
- 返回值为 0 表示元素具有相同的排序优先级。

sortOn() 方法是为具有包含对象的元素的索引数组设计的。这些对象应至少具有一个可用作排序键的公共属性。如果将 sortOn() 方法用于任何其它类型的数组，则会产生意外结果。

以下示例修改 poets 数组，以使每个元素均为对象而非字符串。每个对象既包含诗人的姓又包含诗人的出生年份。

```
var poets:Array = new Array();
poets.push({name:"Angelou", born:"1928"});
poets.push({name:"Blake", born:"1757"});
poets.push({name:"cummings", born:"1894"});
poets.push({name:"Dante", born:"1265"});
poets.push({name:"Wang", born:"701"});
```

可以使用 sortOn() 方法，按照 born 属性对数组进行排序。sortOn() 方法定义两个参数 fieldName 和 options。必须将 fieldName 参数指定为字符串。在以下示例中，使用两个参数 "born" 和 Array.NUMERIC 来调用 sortOn()。Array.NUMERIC 参数用于确保按照数字顺序进行排序，而不是按照字母顺序。即使所有数字具有相同的数位，这也是一种很好的做法，因为当后来在数组中添加较少数位或较多数位的数字时，它会确保排序如期继续进行。

```
poets.sortOn("born", Array.NUMERIC);
for (var i:int = 0; i < poets.length; ++i)
{
  trace(poets[i].name, poets[i].born);
}
/* 输出:
Wang 701
Dante 1265
Blake 1757
cummings 1894
Angelou 1928
*/
```

通常，sort() 和 sortOn() 方法用来修改数组。如果要对数组排序而又不修改现有数组，请将 Array.RETURNINDEXEDARRAY 常量作为 options 参数的一部分进行传递。此选项将指示方法返回反映排序的新数组同时保留原始数组原封不动。方法返回的数组为由反映新排序顺序的索引号组成的简单数组，不包含原始数组的任何元素。例如，如果要根据出生年份对 poets 数组排序，同时不对数组进行修改，请在传递的 options 参数中包括 Array.RETURNINDEXEDARRAY 常量。

以下示例将返回的索引信息存储在名为 indices 的数组中，然后使用 indices 数组和未修改的 poets 数组按出生年份输出诗人：

```
var indices:Array;
indices = poets.sortOn("born", Array.NUMERIC | Array.RETURNINDEXEDARRAY);
for (var i:int = 0; i < indices.length; ++i)
{
  var index:int = indices[i];
  trace(poets[index].name, poets[index].born);
}
/* 输出:
Wang 701
Dante 1265
Blake 1757
cummings 1894
Angelou 1928
*/
```

## 查询数组

Array 类中的其余四种方法 concat()、join()、slice() 和 toString() 用于查询数组中的信息，而不修改数组。concat() 和 slice() 方法返回新数组；而 join() 和 toString() 方法返回字符串。concat() 方法将新数组和元素列表作为参数，并将其与现有数组结合起来创建新数组。 slice() 方法具有两个名为 startIndex 和 endIndex 的参数，并返回一个新数组，它包含从现有数组分离出来的元素副本。分离从 startIndex 处的元素开始，到 endIndex 处的前一个元素结束。值得强调的是， endIndex 处的元素不包括在返回值中。

以下示例通过 concat() 和 slice() 方法，使用其它数组的元素创建一个新数组：

```
var array1:Array = ["alpha", "beta"];
var array2:Array = array1.concat("gamma", "delta");
trace(array2); // 输出: alpha,beta,gamma,delta

var array3:Array = array1.concat(array2);
trace(array3); // 输出: alpha,beta,alpha,beta,gamma,delta

var array4:Array = array3.slice(2,5);
trace(array4); // 输出: alpha,beta,gamma
```

可以使用 join() 和 toString() 方法查询数组，并将其内容作为字符串返回。如果 join() 方法没有使用参数，则这两个方法的行为相同，它们都返回包含数组中所有元素的列表（以逗号分隔）的字符串。与 toString() 方法不同，join() 方法接受名为 delimiter 的参数；可以使用此参数，选择要用作返回字符串中各个元素之间分隔符的符号。

以下示例创建名为 rivers 的数组，并调用 join() 和 toString() 以便按字符串形式返回数组中的值。toString() 方法用于返回以逗号分隔的值 (riverCSV)；而 join() 方法用于返回以 + 字符分隔的值。

```
var rivers:Array = ["Nile", "Amazon", "Yangtze", "Mississippi"];
var riverCSV:String = rivers.toString();
trace(riverCSV); // 输出: Nile,Amazon,Yangtze,Mississippi
var riverPSV:String = rivers.join("+");
trace(riverPSV); // 输出: Nile+Amazon+Yangtze+Mississippi
```

对于 join() 方法，应注意的一个问题是，无论为主数组元素指定的分隔符是什么，为嵌套数组返回的值始终以逗号作为分隔符，如以下示例所示：

```
var nested:Array = ["b","c","d"];
var letters:Array = ["a",nested,"e"];
var joined:String = letters.join("+");
trace(joined); // 输出: a+b,c,d+e
```

# 关联数组

关联数组有时候也称为 “哈希” 或 “映射”，它使用 “键” 而非数字索引来组织存储的值。关联数组中的每个键都是用于访问一个存储值的唯一字符串。关联数组为 Object 类的实例，也就是说每个键都与一个属性名称对应。关联数组是键和值对的无序集合。在代码中，不应期望关联数组的键按特定的顺序排列。

ActionScript 3.0 中引入了名为“字典”的高级关联数组。字典是 flash.utils 包中 Dictionary 类的实例，使用的键可以为任意数据类型，但通常为 Object 类的实例。换言之，字典的键不局限于 String 类型的值。

本部分说明如何创建使用字符串作为键的数组以及如何使用 Dictionary 类。

## 具有字符串键的关联数组

在 ActionScript 3.0 中有两种创建关联数组的方法。第一种方法是使用 Object 构造函数，它的优点是可以使用对象文本初始化数组。 Object 类的实例 （也称作 “通用对象”）在功能上等同于关联数组。通用对象的每个属性名称都用作键，提供对存储的值的访问。

以下示例创建一个名为 `monitorInfo` 的关联数组，并使用对象文本初始化具有两个键和值对的数组：

```
var monitorInfo:Object = {type:"Flat Panel", resolution:"1600 x 1200"};
trace(monitorInfo["type"], monitorInfo["resolution"]);
// 输出: Flat Panel 1600 x 1200
```

如果在声明数组时不需要初始化，可以使用 Object 构造函数创建数组，如下所示：

```
var monitorInfo:Object = new Object();
```

使用对象文本或 Object 类构造函数创建数组后，可以使用括号运算符 (`[]`) 或点运算符 (.) 在数组中添加值。以下示例将两个新值添加到 `monitorArray` 中：

```
monitorInfo["aspect ratio"] = "16:10"; // 格式错误，请勿使用空格
monitorInfo.colors = "16.7 million";
trace(monitorInfo["aspect ratio"], monitorInfo.colors);
// 输出: 16:10 16.7 million
```

请注意，名为 `aspect ratio` 的键包含空格字符。也就是说，空格字符可以与括号运算符一起使用，但试图与点运算符一起使用时会生成一个错误。不建议在键名称中使用空格。

第二种关联数组创建方法是使用 Array 构造函数，然后使用括号运算符 ([]) 或点运算符 (.) 将键和值对添加到数组中。如果将关联数组声明为 Array 类型，则将无法使用对象文本初始化该数组。以下示例使用 Array 构造函数创建一个名为 monitorInfo 的关联数组，并添加一个名为 type 的键和一个名为 resolution 的键以及它们的值：

```
var monitorInfo:Array = new Array();
monitorInfo["type"] = "Flat Panel";
monitorInfo["resolution"] = "1600 x 1200";
trace(monitorInfo["type"], monitorInfo["resolution"]);
// 输出： Flat Panel 1600 x 1200
```

使用 Array 构造函数创建关联数组没有什么优势。即使使用 Array 构造函数或 Array 数据类型，也不能将 Array 类的 Array.length 属性或任何方法用于关联数组。最好将 Array 构造函数用于创建索引数组。

## 具有对象键的关联数组

可以使用 Dictionary 类创建使用对象而非字符串作为键的关联数组。这样的数组有时候也称作字典、哈希或映射。例如，考虑这样一个应用程序，它可根据 Sprite 对象与特定容器的关联确定 Sprite 对象的位置。可以使用 Dictionary 对象，将每个 Sprite 对象映射到一个容器。

以下代码创建三个用作 Dictionary 对象的键的 Sprite 对象实例。它为每个键分配了值 GroupA 或 GroupB。值可以是任意数据类型，但在此示例中，GroupA 和 GroupB 均为 Object 类的实例。然后，可以使用属性访问运算符 ([]) 访问与每个键关联的值，如以下代码所示：

```
import flash.display.Sprite;
import flash.utils.Dictionary;

var groupMap:Dictionary = new Dictionary();

// 要用作键的对象
var spr1:Sprite = new Sprite();
var spr2:Sprite = new Sprite();
var spr3:Sprite = new Sprite();

// 要用作值的对象
var groupA:Object = new Object();
var groupB:Object = new Object();

// 在字典中创建新的键 - 值对。
groupMap[spr1] = groupA;
groupMap[spr2] = groupB;
groupMap[spr3] = groupB;

if (groupMap[spr1] == groupA)
{
  trace("spr1 is in groupA");
}
```

```
if (groupMap[spr2] == groupB)
{
  trace("spr2 is in groupB");
}
if (groupMap[spr3] == groupB)
{
  trace("spr3 is in groupB");
}
```

## 使用对象键循环访问

可以使用 `for..in` 循环或 `for each..in` 循环来循环访问 Dictionary 对象的内容。`for..in` 循环用来基于键进行循环访问；而 `for each..in` 循环用来基于与每个键关联的值进行循环访问。

可以使用 `for..in` 循环直接访问 Dictionary 对象的对象键。还可以使用属性访问运算符 (`[]`) 访问 Dictionary 对象的值。以下代码使用前面的 `groupMap` 字典示例来说明如何使用 `for..in` 循环来循环访问 Dictionary 对象：

```
for (var key:Object in groupMap)
{
  trace(key, groupMap[key]);
}
/* 输出:
[object Sprite] [object Object]
[object Sprite] [object Object]
[object Sprite] [object Object]
*/
```

可以使用 `for each..in` 循环直接访问 Dictionary 对象的值。以下代码也使用 `groupMap` 字典来说明如何使用 `for each..in` 循环来循环访问 Dictionary 对象：

```
for each (var item:Object in groupMap)
{
  trace(item);
}
/* 输出:
[object Object]
[object Object]
[object Object]
*/
```

### 对象键和内存管理

Adobe Flash Player 使用垃圾回收系统来恢复不再使用的内存。当对象不具有指向它的引用时，即可对其进行垃圾回收，并会在下次执行垃圾回收系统时恢复内存。例如，以下代码创建一个新对象，并将对此对象的引用分配给变量 myObject：

```
var myObject:Object = new Object();
```

只要有对此对象的引用，垃圾回收系统就不会恢复此对象占用的内存。如果更改 myObject 的值以使其指向其它对象或将其设置为值 null，并且没有对原始对象的其它引用，则可以对原始对象占用的内存进行垃圾回收。

如果将 myObject 用作 Dictionary 对象中的键，则会创建对原始对象的另一个引用。例如，以下代码创建两个对象引用 （myObject 变量和 myMap 对象中的键）：

```
import flash.utils.Dictionary;

var myObject:Object = new Object();
var myMap:Dictionary = new Dictionary();
myMap[myObject] = "foo";
```

要使 myObject 引用的对象能够进行垃圾回收，您必须删除对它的所有引用。在此情况下，必须更改 myObject 的值并从 myMap 中删除 myObject 键，如以下代码所示：

```
myObject = null;
delete myMap[myObject];
```

此外，它可以使用 Dictionary 构造函数的 useWeakReference 参数，以使所有字典的键成为“弱引用”。垃圾回收系统忽略弱引用，也就是说只具有弱引用的对象可以进行垃圾回收。例如，在以下代码中，要使对象能够进行垃圾回收，您不需要从 myMap 中删除 myObject 键：

```
import flash.utils.Dictionary;

var myObject:Object = new Object();
var myMap:Dictionary = new Dictionary(true);
myMap[myObject] = "foo";
myObject = null; // 使对象能够进行垃圾回收。
```

# 多维数组

多维数组将其它数组作为其元素。例如，考虑一个任务列表，它存储为字符串索引数组：

```
var tasks:Array = ["wash dishes", "take out trash"];
```

如果要将一周中每天的任务存储为一个单独的列表，可以创建一个多维数组，一周中的每天使用一个元素。每个元素包含一个与 tasks 数组类似的索引数组，而该索引数组存储任务列表。在多维数组中，可以使用任意组合的索引数组和关联数组。以下部分中的示例使用了两个索引数组或由索引数组组成的关联数组。练习的时候，您也许需要采用其它组合。

## 两个索引数组

使用两个索引数组时，可以将结果呈现为表或电子表格。第一个数组的元素表示表的行，第二个数组的元素表示表的列。

例如，以下多维数组使用两个索引数组跟踪一周中每一天的任务列表。第一个数组 `masterTaskList` 是使用 Array 类构造函数创建的。此数组中的各个元素分别表示一周中的各天，其中索引 0 表示星期一，索引 6 表示星期日。可将这些元素当成是表的行。可通过为 `masterTaskList` 数组中创建的七个元素中的每个元素分配数组文本来创建每一天的任务列表。这些数组文本表示表的列。

```
var masterTaskList:Array = new Array();
masterTaskList[0] = ["wash dishes", "take out trash"];
masterTaskList[1] = ["wash dishes", "pay bills"];
masterTaskList[2] = ["wash dishes", "dentist", "wash dog"];
masterTaskList[3] = ["wash dishes"];
masterTaskList[4] = ["wash dishes", "clean house"];
masterTaskList[5] = ["wash dishes", "wash car", "pay rent"];
masterTaskList[6] = ["mow lawn", "fix chair"];
```

可以使用括号记号访问任意任务列表中的单个项。第一组括号表示一周的某一天，第二组括号表示这一天的任务列表。例如，要检索星期三的列表中的第二项任务，请首先使用表示星期三的索引 2，然后使用表示列表中的第二项任务的索引 1。

```
trace(masterTaskList[2][1]); // 输出: dentist
```

要检索星期日的任务列表中的第一项，请使用表示星期日的索引 6 和表示列表中的第一项任务的索引 0。

```
trace(masterTaskList[6][0]); // 输出: mow lawn
```

## 具有索引数组的关联数组

要使单个数组的访问更加方便，可以使用关联数组表示一周的各天并使用索引数组表示任务列表。通过使用关联数组可以在引用一周中特定的一天时使用点语法，但要访问关联数组的每个元素还需额外进行运行时处理。以下示例使用关联数组作为任务列表的基础，并使用键和值对来表示一周中的每一天：

```
var masterTaskList:Object = new Object();
masterTaskList["Monday"] = ["wash dishes", "take out trash"];
masterTaskList["Tuesday"] = ["wash dishes", "pay bills"];
masterTaskList["Wednesday"] = ["wash dishes", "dentist", "wash dog"];
masterTaskList["Thursday"] = ["wash dishes"];
masterTaskList["Friday"] = ["wash dishes", "clean house"];
masterTaskList["Saturday"] = ["wash dishes", "wash car", "pay rent"];
masterTaskList["Sunday"] = ["mow lawn", "fix chair"];
```

点语法通过避免使用多组括号改善了代码的可读性。

```
trace(masterTaskList.Wednesday[1]); // 输出: dentist
trace(masterTaskList.Sunday[0]);    // 输出: mow lawn
```

可以使用 for..in 循环来循环访问任务列表，但必须使用括号记号来访问与每个键关联的值，而不是使用点语法。由于 masterTaskList 为关联数组，因而不一定会按照您所期望的顺序检索元素，如以下示例所示：

```
for (var day:String in masterTaskList)
{
  trace(day + ": " + masterTaskList[day])
}
/* output:
Sunday: mow lawn,fix chair
Wednesday: wash dishes,dentist,wash dog
Friday: wash dishes,clean house
Thursday: wash dishes
Monday: wash dishes,take out trash
Saturday: wash dishes,wash car,pay rent
Tuesday: wash dishes,pay bills
*/
```

# 克隆数组

Array 类不具有复制数组的内置方法。可以通过调用不带参数的 concat() 或 slice() 方法来创建数组的 "浅副本"。在浅副本中，如果原始数组具有对象元素，则仅复制指向对象的引用而非对象本身。与原始数组一样，副本也指向相同的对象。对对象所做的任何更改都会在两个数组中反映出来。

在 "深副本" 中，将复制原始数组中的所有对象，从而使新数组和原始数组指向不同的对象。深度复制需要多行代码，通常需要创建函数。可以将此类函数作为全局实用程序函数或 Array 子类的方法来进行创建。

以下示例定义一个名为 clone() 的函数以执行深度复制。其算法采用了一般的 Java 编程技巧。此函数创建深副本的方法是：将数组序列化为 ByteArray 类的实例，然后将此数组读回到新数组中。此函数接受对象，因此既可以将此函数用于索引数组，又可以将其用于关联数组，如以下代码所示：

```
import flash.utils.ByteArray;

function clone(source:Object):*
{
  var myBA:ByteArray = new ByteArray();
  myBA.writeObject(source);
  myBA.position = 0;
  return(myBA.readObject());
}
```

# 高级主题

## 扩展 Array 类

Array 类是少数不是最终类的核心类之一，也就是说您可以创建自己的 Array 子类。本部分提供了创建 Array 子类的示例，并讨论了在创建子类过程中会出现的一些问题。

如前所述，ActionScript 中的数组是不能指定数据类型的，但您可以创建只接受具有指定数据类型的元素的 Array 子类。以下部分中的示例定义名为 TypedArray 的 Array 子类，该子类中的元素限定为具有第一个参数中指定的数据类型的值。这里的 TypedArray 类仅用于说明如何扩展 Array 类，并不一定适用于生产，这有以下若干原因。第一，类型检查是在运行时而非编译时进行的。第二，当 TypedArray 方法遇到不匹配时，将忽略不匹配并且不会引发异常；当然，修改此方法使其引发异常也是很简单的。第三，此类无法防止使用数组访问运算符将任一类型的值插入到数组中。第四，其编码风格倾向于简洁而非性能优化。

### 声明子类

可以使用 `extends` 关键字来指示类为 Array 的子类。与 Array 类一样，Array 的子类应使用 `dynamic` 属性。否则，子类将无法正常发挥作用。

以下代码显示 TypedArray 类的定义，该类包含一个保存数据类型的常量、一个构造函数方法和四个能够将元素添加到数组的方法。此示例省略了各方法的代码，但在以后的部分中将列出这些代码并详加说明：

```
public dynamic class TypedArray extends Array
{
  private const dataType:Class;

  public function TypedArray(...args) {}

  AS3 override function concat(...args):Array {}

  AS3 override function push(...args):uint {}

  AS3 override function splice(...args) {}

  AS3 override function unshift(...args):uint {}
}
```

由于本示例中假定将编译器选项 -as3 设置为 true，而将编译器选项 -es 设置为 false，因而这四个被覆盖的方法均使用 AS3 命名空间而非 public 属性。这些是 Adobe Flex Builder 2 和 Adobe Flash CS3 Professional 的默认设置。有关详细信息，请参阅第 148 页的 “AS3 命名空间”。

> **提示** 如果您是倾向于使用原型继承的高级开发人员，您可能会在以下两个方面对 TypedArray 类进行较小的改动，以使其在编译器选项 -es 设置为 true 的情况下进行编译。一方面，删除出现的所有 override 属性，并使用 public 属性替换 AS3 命名空间。另一方面，使用 Array.prototype 替换出现的所有四处 super。

## TypedArray 构造函数

由于此子类构造函数必须接受一列任意长度的参数，从而造成了一种有趣的挑战。该挑战就是如何将参数传递给超类构造函数以创建数组。如果将一列参数作为数组进行传递，则超类构造函数会将其视为 Array 类型的一个参数，并且生成的数组长度始终只有 1 个元素。传递参数列表的传统处理方式是使用 Function.apply() 方法，此方法将参数数组作为第二个参数，但在执行函数时将其转换为一列参数。遗憾的是， Function.apply() 方法不能和构造函数一起使用。

剩下的唯一方法是在 TypedArray 构造函数中重新创建 Array 构造函数的逻辑。以下代码说明在 Array 类构造函数中使用的算法，您可以在 Array 子类构造函数中重复使用此算法：

```
public dynamic class Array
{
  public function Array(...args)
  {
    var n:uint = args.length
    if (n == 1 && (args[0] is Number))
    {
      var dlen:Number = args[0];
      var ulen:uint = dlen;
      if (ulen != dlen)
      {
        throw new RangeError("Array index is not a 32-bit unsigned integer
  ("+dlen+")");
      }
      length = ulen;
    }
    else
    {
      length = n;
      for (var i:int=0; i < n; i++)
      {
        this[i] = args[i]
      }
    }
  }
}
```

TypedArray 构造函数与 Array 构造函数中的大部分代码都相同，只在四个地方对代码进行了改动。其一，参数列表中新增了一个必需的 Class 类型参数，使用此参数可以指定数组的数据类型。其二，将传递给构造函数的数据类型分配给 `dataType` 变量。其三，在 `else` 语句中，在 `for` 循环之后为 `length` 属性赋值，以使 `length` 只包括相应类型的参数。其四，`for` 循环的主体使用 `push()` 方法的被覆盖版本，以便仅将正确数据类型的参数添加到数组中。以下示例显示 TypedArray 构造函数：

```
public dynamic class TypedArray extends Array
{
  private var dataType:Class;
  public function TypedArray(typeParam:Class, ...args)
  {
    dataType = typeParam;
    var n:uint = args.length
    if (n == 1 && (args[0] is Number))
    {
      var dlen:Number = args[0];
      var ulen:uint = dlen
      if (ulen != dlen)
      {
        throw new RangeError("Array index is not a 32-bit unsigned integer
  ("+dlen+")")
      }
      length = ulen;
    }
    else
    {
      for (var i:int=0; i < n; i++)
      {
        // 在 push() 中完成类型检查
        this.push(args[i])
      }
      length = this.length;
    }
  }
}
```

## TypedArray 覆盖的方法

TypedArray 类覆盖前述四种能够将元素添加到数组的方法。在每种情况下，被覆盖方法均添加类型检查，这种检查可以防止添加不正确数据类型的元素。然后，每种方法均调用其自身的超类版本。

push() 方法使用 for..in 循环来循环访问参数列表，并对每个参数执行类型检查。可以使用 splice() 方法，从 args 数组中删除所有不是正确类型的参数。在 for..in 循环结束之后，args 数组中将只包含类型为 dataType 的值。然后对更新后的 args 数组调用 push() 的超类版本，如以下代码所示：

```
AS3 override function push(...args):uint
{
    for (var i:* in args)
    {
        if (!(args[i] is dataType))
        {
            args.splice(i,1);
        }
    }
    return (super.push.apply(this, args));
}
```

concat() 方法创建一个名为 passArgs 的临时 TypedArray 来存储通过类型检查的参数。这样，便可以重复使用 push() 方法中的类型检查代码。for..in 循环用于循环访问 args 数组，并为每个参数调用 push()。由于将 passArgs 指定为类型 TypedArray，因而将执行 push() 的 TypedArray 版本。然后，concat() 方法将调用其自身的超类版本，如以下代码所示：

```
AS3 override function concat(...args):Array
{
    var passArgs:TypedArray = new TypedArray(dataType);
    for (var i:* in args)
    {
        // 在 push() 中完成类型检查
        passArgs.push(args[i]);
    }
    return (super.concat.apply(this, passArgs));
}
```

splice() 方法使用任意一列参数，但前两个参数始终指的是索引号和要删除的元素个数。这就是为什么被覆盖的 splice() 方法仅对索引位置 2 或其以后的 args 数组元素执行类型检查。该代码中一个有趣的地方是，for 循环中的 splice() 调用看上去似乎是递归调用，但实际上并不是递归调用，这是由于 args 的类型是 Array 而非 TypedArray，也就是说，args.splice() 调用是对此方法超类版本的调用。在 for..in 循环结束后，args 数组中将只包含索引位置 2 或其以后位置具有正确类型的值，并且 splice() 会调用其自身的超类版本，如以下代码所示：

```
    AS3 override function splice(...args):*
    {
        if (args.length > 2)
        {
            for (var i:int=2; i< args.length; i++)
            {
                if (!(args[i] is dataType))
                {
                    args.splice(i,1);
                }
            }
        }
        return (super.splice.apply(this, args));
    }
```

用于将元素添加到数组开头的 unshift() 方法也可以接受任意一列参数。被覆盖的 unshift() 方法使用的算法与 push() 方法使用的算法类似，如以下示例代码所示：

```
    AS3 override function unshift(...args):uint
    {
        for (var i:* in args)
        {
            if (!(args[i] is dataType))
            {
                args.splice(i,1);
            }
        }
        return (super.unshift.apply(this, args));
    }
}
```

# 示例：PlayList

此 PlayList 示例在管理歌曲列表的音乐播放列表应用程序环境中展示了使用数组的多种技巧。这些方法包括：

- 创建索引数组
- 向索引数组中添加项
- 使用不同的排序选项按照不同的属性对对象数组排序
- 将数组转换为以字符分隔的字符串

要获取该范例的应用程序文件，请访问
www.adobe.com/go/learn_programmingAS3samples_flash_cn。
可以在 Samples/PlayList 文件夹中找到 PlayList 应用程序文件。该应用程序包含以下文件：

| 文件 | 说明 |
|---|---|
| PlayList.mxml<br>或<br>PlayList.fla | Flash 或 Flex 中的主应用程序文件 （分别为 FLA 和 MXML）。 |
| com/example/programmingas3/playlist/Song.as | 表示一首歌曲信息的值对象。由 PlayList 类管理的项为 Song 实例。 |
| com/example/programmingas3/playlist/SortProperty.as | 伪枚举，其可用值代表 Song 类的属性，可以根据这些属性对 Song 对象列表排序。 |

## PlayList 类概述

PlayList 类管理一组 Song 对象。它具有一些公共方法，可将歌曲添加到播放列表（addSong() 方法）以及对列表中的歌曲排序 （sortList() 方法） 。此外，它还包括一个只读存取器属性 songList，可提供对播放列表中实际歌曲集的访问。在内部，PlayList 类使用私有 Array 变量跟踪其歌曲：

```
public class PlayList
{
   private var _songs:Array;
   private var _currentSort:SortProperty = null;
   private var _needToSort:Boolean = false;
   ...
}
```

除了 PlayList 类用来跟踪其歌曲列表的 _songs Array 变量以外，还有另外两个私有变量，分别用来跟踪是否需要对列表排序 (_needToSort) 以及在给定时间对列表进行排序所依据的属性 (_currentSort)。

跟所有对象一样，声明 Array 实例只做了创建 Array 工作的一半。在访问 Array 实例的属性或方法之前，必须在 PlayList 类的构造函数中完成对它的实例化。

```
public function PlayList()
{
    this._songs = new Array();
    // 设置初始排序。
    this.sortList(SortProperty.TITLE);
}
```

构造函数的第一行用于实例化 _songs 变量，以使其处于待用状态。此外，还会调用 sortList() 方法来设置初始排序所依据的属性。

## 向列表中添加歌曲

用户在应用程序中输入新歌曲后，数据条目表单上的代码将调用 PlayList 类的 addSong() 方法。

```
/**
 * 向播放列表中添加歌曲。
 */
public function addSong(song:Song):void
{
    this._songs.push(song);
    this._needToSort = true;
}
```

在 addSong() 内，将调用 _songs 数组的 push() 方法，以便将传递给 addSong() 的 Song 对象添加为该数组中的新元素。不管之前应用的是哪种排序方法，使用 push() 方法时，都会将新元素添加到数组末尾。也就是说，调用 push() 方法后，歌曲列表的排序很可能不正确，因此将 _needToSort 变量设置为 true。理论上说，可以立即调用 sortList() 方法，而无需跟踪是否要在给定时间对列表进行排序。实际上，不需要立即对歌曲列表排序，除非要对其进行检索。通过延迟排序操作，应用程序不会执行不必要的排序，例如，在将几首歌曲添加到列表中且不需要对其进行检索时。

## 对歌曲列表排序

由于由 PlayList 管理的 Song 实例为复杂对象，因而此应用程序的用户可能要根据不同的属性（例如，歌曲名称或发行年份）来对播放列表排序。在 PlayList 应用程序中，对歌曲列表排序的任务由以下三部分组成：确定列表排序所依据的属性，指出按照该属性排序时应使用的排序操作，以及执行实际排序操作。

## 排序属性

Song 对象跟踪若干属性，包括歌曲名称、歌手、发行年份、文件名和用户选择的歌曲所属流派。在这些属性当中，只有前三个可用于排序。为了方便开发人员，此示例中包括 SortProperty 类，此类与枚举作用相同，其值表示可用于排序的属性。

```
public static const TITLE:SortProperty = new SortProperty("title");
public static const ARTIST:SortProperty = new SortProperty("artist");
public static const YEAR:SortProperty = new SortProperty("year");
```

SortProperty 类包含 TITLE、ARTIST 和 YEAR 三个常量，其中的每个常量都存储一个包含相关 Song 类属性（可用于排序）的实际名称的字符串。在代码的其余部分，只要指出排序属性，都是使用枚举成员完成的。例如，在 PlayList 构造函数中，最初通过调用 sortList() 方法对列表排序，如下所示：

```
    // 设置初始排序。
    this.sortList(SortProperty.TITLE);
```

由于将排序属性指定为 SortProperty.TITLE，因而将根据歌曲名称对歌曲排序。

## 依据属性排序和指定排序选项

对歌曲列表排序的实际工作是由 PlayList 类在 sortList() 方法中进行的，如下所示：

```
/**
 * 依据指定属性对歌曲列表排序。
 */
public function sortList(sortProperty:SortProperty):void
{
    ...
    var sortOptions:uint;
    switch (sortProperty)
    {
        case SortProperty.TITLE:
            sortOptions = Array.CASEINSENSITIVE;
            break;
        case SortProperty.ARTIST:
            sortOptions = Array.CASEINSENSITIVE;
            break;
        case SortProperty.YEAR:
            sortOptions = Array.NUMERIC;
            break;
    }

    // 对数据执行实际的排序操作。
    this._songs.sortOn(sortProperty.propertyName, sortOptions);

    // 保存当前排序属性。
    this._currentSort = sortProperty;

    // 记录列表已排序。
    this._needToSort = false;
}
```

如果根据歌名或歌手排序，则应按照字母顺序排序，要根据年份排序，则按照数字顺序排序最为合理。switch 语句用于根据 sortProperty 参数中指定的值定义合适的排序选项，此选项将存储在 sortOptions 变量中。这里，再次使用命名的枚举成员而非硬编码值来区分不同属性。

确定排序属性和排序选项之后，将通过调用 sortOn() 方法并将上述两个值作为参数传递来完成对 _songs 数组的排序。对歌曲列表进行排序之后，记录当前排序属性。

## 将数组元素组合为以字符分隔的字符串

在本示例中，数组除了用于在 PlayList 类中维护歌曲列表以外，还用于在 Song 类中帮助管理指定歌曲所属的流派的列表。请考虑 Song 类定义中的以下代码片断：

```
private var _genres:String;

public function Song(title:String, artist:String, year:uint,
  filename:String, genres:Array)
{
  ...
  // 流派作为数组传递进来
  // 但存储为以分号分隔的字符串。
  this._genres = genres.join(";");
}
```

创建新 Song 实例时，将把用于指定歌曲所属的一个或多个流派的 genres 参数定义为 Array 实例。这有助于将多个流派组合为一个可以传递给构造函数的变量。但在内部，Song 类在 _genres 私有变量中以分号分隔的 String 实例形式来保存流派。通过调用 join() 方法（使用文本字符串值 ";" 作为指定分隔符），Array 参数将转换为以分号分隔的字符串。

通过使用相同的标记，genres 存取器可将流派作为 Array 进行设置或检索：

```
  public function get genres():Array
  {
    // 流派存储为以分号分隔的字符串，
    // 因此需要将它们转换为 Array 以将它们传递出去。
    return this._genres.split(";");
  }
  public function set genres(value:Array):void
  {
    // 流派作为数组传递进来，
    // 但存储为以分号分隔的字符串。
    this._genres = value.join(";");
  }
```

genres set 存取器的行为与构造函数完全相同；它接受 Array 并调用 join() 方法以将其转换为以分号分隔的 String。get 存取器执行相反的操作：调用 _genres 变量的 split() 方法，使用指定分隔符（采用与前面相同的文本字符串值 ";"）将字符串拆分为由值组成的数组。

第 8 章

# 处理错误

8

“处理”错误是指在应用程序中构建用来响应错误或修正错误的逻辑，这些错误在编译应用程序或在运行经过编译的应用程序时产生。如果应用程序能够处理错误，则在遇到错误时，应用程序会执行“一些操作”作为响应，而不是没有任何响应并且引发该错误的进程在没有提示的情况下发生失败。正确使用错误处理有助于防止应用程序和应用程序的使用者执行其它意外行为。

不过，错误处理涵盖的内容很广，它包括对编译期间或运行时引发的许多种错误予以响应。本章重点介绍如何处理运行时错误、可能生成哪些不同类型的错误以及 ActionScript 3.0 中新的错误处理系统的优点。本章还将介绍如何为您的应用程序实现您自己自定义的错误处理策略。

## 目录

# 错误处理基础知识

## 错误处理简介

运行时错误是指阻止在 Adobe Flash Player 中运行 ActionScript 内容的 ActionScript 代码错误。要确保为用户平稳地运行 ActionScript 代码，您必须在应用程序中编写能够处理该错误的代码，即修正该错误，解决该问题，或者至少让用户知道发生了什么错误。此过程称为 “错误处理”。

错误处理涵盖的内容很广，它包括对编译期间或运行时引发的许多种错误予以响应。通常，比较容易找出在编译时发生的错误；您必须修正这些错误才能完成 SWF 文件的创建过程。本章没有讨论编译时错误；有关编写不包含编译时错误的代码的详细信息，请参阅第 55 页的第 3 章 “ActionScript 语言及其语法”和第 115 页的第 4 章 “ActionScript 中面向对象的编程”。本章重点介绍运行时错误。

运行时错误可能更难于检测，因为必须实际运行错误代码才会发生这些错误。如果程序片断包含几个代码分支 （如 `if..then..else` 语句），您必须通过实际用户可能使用的所有可能的输入值来测试每种可能的条件，以便确认代码不包含任何错误。

运行时错误可以分为以下两类：“程序错误”是指 ActionScript 代码中的错误，如为方法参数指定了错误的数据类型； “逻辑错误”是指程序的逻辑 （数据检查和值处理）错误，如在银行业应用程序中使用错误的公式来计算利率。同样，通过事先仔细地测试应用程序，通常可以检测到并纠正这两种类型的错误。

理想情况下，您希望在将应用程序发布到最终用户之前找出并消除其中的所有错误。但是，并非所有错误都是可以预见或避免的。例如，假设 ActionScript 应用程序从特定网站加载信息，而您无法控制该网站。如果该网站在某一时刻不可用，则依赖于该外部数据的应用程序部分将无法正确运行。错误处理的最重要方面涉及为这些未知情况做好准备并妥善处理它们，以使用户能够继续使用应用程序，或者至少收到一条友好的错误消息以解释无法工作的原因。

ActionScript 使用以下两种方式来表示运行时错误：

- 错误类：很多错误都具有一个关联的错误类。当发生错误时，Flash Player 将创建与该错误关联的特定错误类的实例。代码可以使用该错误对象中包含的信息对错误进行相应的响应。
- 错误事件：有时，当 Flash Player 正常触发事件时，也会发生错误。在这些情况下，Flash Player 触发的是错误事件。与其它事件一样，每个错误事件都具有一个关联的类，Flash Player 会将该类的实例传递给订阅了该错误事件的方法。

要确定特定方法能否触发错误或错误事件，请参阅 《ActionScript 3.0 语言和组件参考》中的方法条目。

## 常见错误处理任务

您需要对代码执行的与错误处理相关的常见任务：

- 编写代码以处理错误
- 测试、捕获以及重新引发错误
- 定义您自己的错误类
- 响应错误和状态事件

## 重要概念和术语

以下参考列表包含将会在本章中遇到的重要术语：

- 异步 (Asynchronous)：不提供即时结果的程序命令（如方法调用），而以事件形式提供结果（或错误）。
- 捕获 (Catch)：如果发生了异常（运行时错误）并且代码注意到该异常，则认为该代码"捕获"了异常。捕获异常后， Flash Player 将停止通知其它 ActionScript 代码发生了异常。
- 调试版 (Debugger version)：一种特殊的 Flash Player 版本，其中包含用于通知用户发生了运行时错误的代码。在标准 Flash Player 版本（大多数用户使用的版本）中，Flash Player 忽略 ActionScript 代码没有处理的错误。在调试版（Adobe Flash CS3 Professional 和 Adobe Flex 附带提供）中，当发生未处理的错误时，将显示一条警告消息。
- 异常 (Exception)：在程序运行时发生的错误，并且运行时环境（即 Flash Player）无法自行解决该问题。
- 重新引发 (Re-throw): 当代码捕获异常时，Flash Player 不再通知其它对象发生了异常。如果通知其它对象发生了异常非常重要，则代码必须"重新引发"异常以再次启动通知过程。
- 同步 (Synchronous)：提供即时结果或立即引发错误的程序命令（如方法调用），这意味着可以在相同的代码块内使用响应。
- 引发 (Throw)：通知 Flash Player 发生了错误（因而通知了其它对象和 ActionScript 代码）的操作称为"引发"错误。

## 完成本章中的示例

学习本章的过程中，您可能想要自己动手测试一些示例代码清单。实质上本章中的代码清单包括适当的 `trace()` 函数调用。要测试本章中的代码清单，请执行以下操作：

1. 创建一个空的 Flash 文档。
2. 在时间轴上选择一个关键帧。
3. 打开 “动作” 面板，将代码清单复制到 “脚本” 窗格中。
4. 使用 “控制” > “测试影片” 运行程序。

   您将在 “输出” 面板中看到该代码清单的 `trace` 函数的结果。

后面的某些代码清单更为复杂一些，并且是以类的形式编写的。要测试这些示例，请执行以下操作：

1. 创建一个空的 Flash 文档并将它保存到您的计算机上。
2. 创建一个新的 ActionScript 文件，并将它保存到 Flash 文档所在的目录中。文件名应与代码清单中的类的名称一致。例如，如果代码清单定义一个名为 ErrorTest 的类，请使用名称 ErrorTest.as 来保存 ActionScript 文件。
3. 将代码清单复制到 ActionScript 文件中并保存该文件。
4. 在 Flash 文档中，单击舞台或工作区的空白部分，以激活文档的 “属性” 检查器。
5. 在 “属性” 检查器的 “文档类” 字段中，输入您从文本中复制的 ActionScript 类的名称。
6. 使用 “控制” > “测试影片” 运行程序。

   您将在 “输出” 面板中（如果示例使用 `trace()` 函数）或在由示例代码创建的文本字段中看到示例的结果。

测试示例代码清单的这些技术将在第 53 页的 “测试本章内的示例代码清单”中加以详细说明。

# 错误类型

开发和运行应用程序时，您会遇到不同类型的错误和错误术语。下面列出了主要的错误类型和术语：

- *编译时错误*，这类错误在代码编译期间由 ActionScript 编译器引发。当代码中的语法问题导致应用程序无法生成时即会发生编译时错误。
- *运行时错误*，这类错误在应用程序编译之后运行时发生。运行时错误指在 Adobe Flash Player 9 中播放 SWF 文件时产生的错误。大多数情况下，您都可以在运行时错误发生时对其进行处理，将错误报告给用户，并采取相应的步骤让应用程序继续运行。如果是致命错误，如未能连接到远程 Web 站点或未能加载需要的数据，则可以使用错误处理让应用程序妥善地终止运行。

- *同步错误*，这类错误是在调用某个函数时发生的运行时错误。例如，当试图使用某个特定方法但传递给该方法的参数无效时，Flash Player 就会引发异常。多数错误都是在语句执行时同步发生，并且控制流会立即传递给最适用的 catch 语句。

  例如，以下代码将会引发一个运行时错误，原因是在程序试图上载文件之前没有调用 browse() 方法：

```
var fileRef:FileReference = new FileReference();
try
{
  fileRef.upload("http://www.yourdomain.com/fileupload.cfm");
}
catch (error:IllegalOperationError)
{
  trace(error);
  // 错误 #2037: 函数的调用顺序不正确，或者之前的
  // 调用未成功。
}
```

  在本例中，将同步引发一个运行时错误，原因是 Flash Player 断定在试图上载文件之前没有调用 browse() 方法。

  有关同步错误处理的详细信息，请参阅第 223 页的 “在应用程序中处理同步错误”。

- *异步错误*，这类错误是在应用程序运行期间在不同点处发生的运行时错误，它们会产生相应事件并由事件侦听器捕获。在异步操作中，函数发起操作但并不等待操作完成。您可以创建错误事件侦听器来等待应用程序或用户尝试执行某个操作。如果操作失败，则使用一个事件侦听器捕获错误并响应错误事件。然后，该事件侦听器调用一个事件处理函数，以便通过一种有益的方式来响应错误事件。例如，事件处理函数可以启动一个对话框，以提示用户解决该错误。

  下面以前面提到的文件上载同步错误为例。如果在文件上载开始之前成功调用了 browse() 方法，则 Flash Player 会调度若干个事件。例如，上载开始时，将调度 open 事件。文件上载操作成功完成时，将调度 complete 事件。由于事件处理是异步的（即不是在特定、已知或预先指定的时间发生），因此需要使用 addEventListener() 方法来侦听这些特定的事件，如以下代码所示：

```
var fileRef:FileReference = new FileReference();
fileRef.addEventListener(Event.SELECT, selectHandler);
fileRef.addEventListener(Event.OPEN, openHandler);
fileRef.addEventListener(Event.COMPLETE, completeHandler);
fileRef.browse();

function selectHandler(event:Event):void
{
  trace("...select...");
  var request:URLRequest = new URLRequest("http://www.yourdomain.com/
  fileupload.cfm");
  request.method = URLRequestMethod.POST;
  event.target.upload(request.url);
```

```
}
function openHandler(event:Event):void
{
  trace("...open...");
}
function completeHandler(event:Event):void
{
  trace("...complete...");
}
```

有关异步错误处理的详细信息，请参阅第 229 页的 “响应错误事件和状态”。

- *未捕获的异常*，这类错误在引发后并没有相应的逻辑（如 `catch` 语句）来响应它。应用程序引发错误后，如果在当前级别或更高级别找不到适当的 `catch` 语句或事件处理函数来处理错误，则认为该错误是未捕获的异常。

  鉴于用户不一定能够自行解决错误，如果错误并未使当前 SWF 文件停止播放，按照设计， Flash Player 在运行时将忽略未捕获的异常并尝试继续播放该文件。忽略未捕获异常的过程称为 “无提示失败”，它会使应用程序的调试变得非常复杂。 Flash Player 的调试版对未捕获异常的响应方式是终止当前脚本并在 `trace` 语句输出中显示未捕获的异常或将错误消息写入日志文件。如果异常对象是 Error 类或其子类的一个实例，则将调用 `getStackTrace()` 方法，并且还会在 trace 语句输出或日志文件中显示堆栈跟踪信息。有关使用 Flash Player 调试版的详细信息，请参阅第 222 页的 “处理 Flash Player 的调试版”。

# ActionScript 3.0 中的错误处理

由于许多应用程序在没有构建错误处理逻辑的情况下也可以运行，因此开发人员经常将在应用程序中构建错误处理逻辑的工作放在次要地位。但如果没有错误处理逻辑，应用程序容易出现问题而造成非正常停止，或者是用户在应用程序没有按预期工作时而感到莫名其妙。ActionScript 2.0 具有一个 Error 类，可用来在自定义函数中构建逻辑，以便引发具有特定消息的异常。由于错误处理对于构建用户友好的应用程序至关重要，因此 ActionScript 3.0 提供了一个扩展的体系结构用来捕获错误。




尽管 《ActionScript 3.0 语言和组件参考》中介绍了许多方法引发的异常，但可能并未涵盖每种方法可能引发的所有异常。对于某个方法，即使方法描述中确实列出了它可能引发的某些异常，但仍可能存在由于未在方法描述中明确阐述的语法错误或其它问题而引发的异常。

# ActionScript 3.0 错误处理的构成元素

ActionScript 3.0 提供了许多用来进行错误处理的工具，其中包括：

- 错误类。依据 ECMAScript (ECMA-262) 第 4 版语言规范草案，ActionScript 3.0 提供了丰富的错误类，从而扩展了可能产生错误对象的情形的范围。每个错误类都可以帮助应用程序处理和响应特定的错误条件，无论这些错误条件是与系统错误相关（如 MemoryError 条件）、与代码编写错误相关（如 ArgumentError 条件）、与网络和通信错误相关（如 URIError 条件），还是与其它情形相关。有关每个类的详细信息，请参阅第 232 页的 "比较错误类"。
- 更少的无提示失败。在 Flash Player 以前的版本中，只有明确使用了 `throw` 语句，才会产生并报告错误。对于 Flash Player 9，本机的 ActionScript 方法和属性即会引发运行时错误，使用这些方法和属性可以在异常发生时更加有效地处理它们，然后对每个异常予以单独响应。
- 在调试期间显示清楚的错误消息。使用 Flash Player 的调试版时，有问题的代码或情形会生成丰富的错误消息，帮助您轻松识别特定代码块失败的原因。这会使错误的修正工作更为高效。有关详细信息，请参阅第 222 页的 "处理 Flash Player 的调试版"。
- 精确的错误指示可以在运行时向用户显示清楚的错误消息。在 Flash Player 的先前版本中，如果 `upload()` 调用不成功，则 `FileReference.upload()` 方法会返回布尔值 `false`，表示发生了五个可能错误中的一种。在 ActionScript 3.0 中调用 `upload()` 方法时，如果发生错误，您可以引发四种特定错误中的一种，从而有助于向最终用户显示更加精确的错误消息。
- 经过优化完善的错误处理。一些明显的错误是由许多常见原因引发的。例如，在 ActionScript 2.0 中，填充 FileReference 对象之前，`name` 属性的值为 `null`（因此，在使用或显示 `name` 属性之前，需要确保已经设定该值且该值不为 `null`）。在 ActionScript 3.0 中，如果在填充 `name` 属性之前试图访问该属性，Flash Player 将引发 IllegalOperationError，通知您尚未设定该值，这时您可以使用 `try..catch..finally` 块来处理该错误。有关详细信息，请参阅第 223 页的 "使用 try..catch..finally 语句"。
- 没有明显的性能缺点。与 ActionScript 的先前版本相比，使用 `try..catch..finally` 块仅占用很少的额外资源或根本不占用任何额外资源。
- 允许针对特定异步错误事件构建侦听器的 ErrorEvent 类。有关详细信息，请参阅第 229 页的 "响应错误事件和状态"。

## 错误处理策略

只要应用程序未遇到会导致问题的条件，则即使未在程序代码中构建错误处理逻辑，应用程序仍然可以成功运行。但是，如果您没有主动处理错误，并且应用程序确实遇到了问题，用户在应用程序失败时将无从知道原因所在。

在应用程序中进行错误处理有几种不同的方式。下面概括介绍三种主要的错误处理方式：

- 使用 `try..catch..finally` 语句。这些语句可在同步错误发生时捕获它们。可以将语句嵌套在一个层次结构中，以便在不同的代码执行级别捕获异常。有关详细信息，请参阅第 223 页的 "使用 try..catch..finally 语句"。
- 创建您自己的自定义错误对象。您可以使用 Error 类创建您自己的自定义错误对象，以便跟踪应用程序中内置错误类型未涵盖的特定操作。然后，您可以对自定义错误对象使用 `try..catch..finally` 语句。有关详细信息，请参阅第 228 页的"创建自定义错误类"。
- 编写用以响应错误事件的事件侦听器和处理函数。使用此策略，您可以创建全局错误处理函数来处理类似的事件，而无需在 `try..catch..finally` 代码块中复制许多代码。您还可以使用此方法来捕获异步错误。有关详细信息，请参阅第 229 页的 "响应错误事件和状态"。

# 处理 Flash Player 的调试版

Adobe 为开发人员提供了一个特殊版本的 Flash Player 来帮助调试。安装 Adobe Flash CS3 Professional 或 Adobe Flex Builder 2 后，您可以获得一个 Flash Player 调试版副本。

Flash Player 的调试版与发行版在错误指示方面有明显的不同。调试版显示错误类型 （如一般错误、 IOError 或 EOFError）、错误编号和可读错误消息。发行版则仅显示错误类型和错误编号。例如，请考虑使用以下代码：

```
try
{
  tf.text = myByteArray.readBoolean();
}
catch (error:EOFError)
{
  tf.text = error.toString();
}
```

如果 `readBoolean()` 方法在 Flash Player 的调试版中引发 EOFError 异常，则会在 `tf` 文本字段中显示以下消息："EOFError: 错误 #2030: 到达文件末尾。"

同样的代码在 Flash Player 的发行版中则会显示以下文字："EOFError: 错误 #2030。"

为使 Flash Player 的资源和大小在发行版中尽可能地小，因此没有提供错误消息字符串。您可以通过在文档 （《ActionScript 3.0 语言和组件参考》的附录）中查找错误编号来找到相应的错误消息。或者，也可以使用 Flash Player 调试版重现错误以查看完整的错误消息。

# 在应用程序中处理同步错误

最常见的错误处理是同步错误处理逻辑，您可以在处理逻辑中将适当的语句插入代码，以便在运行时捕获同步错误。这种错误处理可以让应用程序在功能失败时注意到发生运行时错误并从错误中恢复。同步错误捕获逻辑中包括 `try..catch..finally` 语句，从字面意义上看，这种方式先尝试 (try) 某个操作，然后捕获 (catch) 来自 Flash Player 的任何错误响应，最后 (finally) 执行另外的操作来处理失败的操作。

## 使用 try..catch..finally 语句

处理同步运行时错误时，可以使用 `try..catch..finally` 语句来捕获错误。当发生运行时错误时， Flash Player 将引发异常，这意味着 Flash Player 将暂停正常的操作而创建一个 Error 类型的特殊对象。 Error 对象随后会被引发到第一个可用的 `catch` 块。

`try` 语句将有可能产生错误的语句括在一起。`catch` 语句应始终与 `try` 语句一起使用。如果在 `try` 语句块的其中某个语句中检测到错误，则会执行附加到该 `try` 语句的 `catch` 语句。

`finally` 语句将无论 `try` 语句块中是否发生错误均要执行的语句括在一起。如果没有错误，`finally` 块中的语句将在 `try` 语句块执行完毕之后执行。如果有错误，则首先执行相应的 `catch` 语句，然后执行 `finally` 块中的语句。

以下代码说明了使用 `try..catch..finally` 语句的语法：

```
try
{
    // 可能会引发错误的一些代码
}
catch (err:Error)
{
    // 用于响应错误的代码
}
finally
{
    // 无论是否引发错误都会运行的代码。此代码可在发生错误之后清除错误，
   // 或者采取措施使应用程序继续运行。
}
```

每个 `catch` 语句识别它要处理的特定类型的异常。`catch` 语句指定的错误类只能是 Error 类的子类。将按顺序检查每个 `catch` 语句。只执行与所引发错误的类型相匹配的第一个 `catch` 语句。换言之，如果首先检查更高一级的 Error 类，然后再检查 Error 类的子类，则仅会匹配更高一级的 Error 类。以下代码说明了这一点：

```
try
{
  throw new ArgumentError("I am an ArgumentError");
}
catch (error:Error)
```

```
{
  trace("<Error> " + error.message);
}
catch (error:ArgumentError)
{
  trace("<ArgumentError> " + error.message);
}
```

上面这段代码的输出如下:

```
<Error> I am an ArgumentError
```

为正确捕获 ArgumentError，需要确保先列出最具体的错误类型，然后再列出较为一般的错误类型，如以下代码所示:

```
try
{
  throw new ArgumentError("I am an ArgumentError");
}
catch (error:ArgumentError)
{
  trace("<ArgumentError> " + error.message);
}
catch (error:Error)
{
  trace("<Error> " + error.message);
}
```

Flash Player API 中有几种方法和属性，如果在执行时它们遇到错误，便会引发运行时错误。例如，Sound 类中的 `close()` 方法，它如果无法关闭音频流，便会引发 IOError 错误，如以下代码所示:

```
var mySound:Sound = new Sound();
try
{
  mySound.close();
}
catch (error:IOError)
{
  // 错误 #2029: 此 URLStream 对象没有打开的流。
}
```

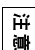
随着您对《ActionScript 3.0 语言和组件参考》的逐渐熟悉，您会注意到哪些方法会引发异常（详见每个方法的说明）。

## throw 语句

如果 Flash Player 在应用程序运行时遇到错误，便会引发异常。此外，您也可以自己使用 throw 语句显式引发异常。如果是显式引发错误，Adobe 建议您引发 Error 类或其子类的实例。以下代码所展示的 throw 语句引发一个 Error 类实例 MyErr，并且最后调用一个函数 myFunction() 在引发错误之后进行响应：

```
var MyError:Error = new Error("Encountered an error with the numUsers value",
  99);
var numUsers:uint = 0;
try
{
  if (numUsers == 0)
  {
    trace("numUsers equals 0");
  }
}
catch (error:uint)
{
  throw MyError; // 捕获无符号整数错误。
}
catch (error:int)
{
  throw MyError; // 捕获整数错误。
}
catch (error:Number)
{
  throw MyError; // 捕获数值错误。
}
catch (error:*)
{
  throw MyError; // 捕获任何其它错误。
}
finally
{
  myFunction(); // 在此执行任何必要的清理工作。
}
```

请注意， catch 语句进行了排序，以便先列出最具体的数据类型。如果先列出的是用于 Number 数据类型的 catch 语句，则无论什么情况下都不会执行用于 uint 数据类型的 catch 语句，也不会执行用于 int 数据类型的 catch 语句。




在 Java 编程语言中，每个可以引发异常的函数都必须先声明这一点，并在附加到函数声明的 throw 子句中列出该函数可以引发的异常类。 ActionScript 不要求您声明函数可以引发的异常。

## 显示简单错误消息

新的异常和错误事件模型的一个最大优点就是：它可以让您向用户告知操作失败的时间和原因。您的工作是编写用来显示消息的代码和在响应中提供选项。

以下代码显示的是一个简单的 `try..catch` 语句，它可以在一个文本字段中显示错误：

```
package
{
  import flash.display.Sprite;
  import flash.text.TextField;

  public class SimpleError extends Sprite
  {
    public var employee:XML =
      <EmpCode>
        <costCenter>1234</costCenter>
        <costCenter>1-234</costCenter>
      </EmpCode>;

    public function SimpleError()
    {
      try
      {
        if (employee.costCenter.length() != 1)
        {
          throw new Error("Error, employee must have exactly one cost
  center assigned.");
        }
      }
      catch (error:Error)
      {
        var errorMessage:TextField = new TextField();
        errorMessage.autoSize = TextFieldAutoSize.LEFT;
        errorMessage.textColor = 0xFF0000;
        errorMessage.text = error.message;
        addChild(errorMessage);
      }
    }
  }
}
```

与 ActionScript 的先前版本相比， ActionScript 3.0 使用了更加丰富的错误类和内置编译器错误，因此可以提供有关失败原因的更多信息。这就使您可以构建更加稳定且具有更佳错误处理能力的应用程序。

## 重新引发错误

构建应用程序时，有时候如果无法正确处理错误，则可能需要重新引发该错误。例如，以下代码显示一个嵌套的 `try..catch` 块，它将在嵌套的 `catch` 块无法处理错误的情况下重新引发一个自定义的 ApplicationError 错误：

```
try
{
  try
  {
    trace("<< try >>");
    throw new ArgumentError("some error which will be rethrown");
  }
  catch (error:ApplicationError)
  {
    trace("<< catch >> " + error);
    trace("<< throw >>");
    throw error;
  }
  catch (error:Error)
  {
    trace("<< Error >> " + error);
  }
}
catch (error:ApplicationError)
{
  trace("<< catch >> " + error);
}
```

上面这个代码片断的输出如下：

```
<< try >>
<< catch >> ApplicationError: some error which will be rethrown
<< throw >>
<< catch >> ApplicationError: some error which will be rethrown
```

嵌套的 `try` 块引发一个自定义的 ApplicationError 错误，该错误由后面的 `catch` 块捕获。此嵌套的 `catch` 块会尝试处理错误，如果不成功，则将 ApplicationError 对象引发到包含此 catch 块的 `try..catch` 块。

# 创建自定义错误类

您可以通过扩展其中一种标准的错误类，在 ActionScript 中创建您自己的专用错误类。有多种原因需要创建您自己的错误类：

- 识别应用程序特有的错误或错误组。

  例如，除了由 Flash Player 捕获的错误外，您可能希望对您自己的代码所引发的错误采取另外的操作。您可以创建 Error 类的一个子类，以便在 `try..catch` 块中跟踪新的错误数据类型。

- 为应用程序生成的错误提供特有的错误显示能力。

  例如，可以创建一个以某种方式设置错误消息格式的新的 `toString()` 方法。还可以定义一个 `lookupErrorString()` 方法，该方法获取错误代码并根据用户的语言首选参数查找适当的消息。

专用的错误类必须扩展 ActionScript 的核心错误类。以下是一个扩展了 Error 类的专用 AppError 类示例：

```
public class AppError extends Error
{
  public function AppError(message:String, errorID:int)
  {
    super(message, errorID);
  }
}
```

以下是在项目中使用 AppError 的一个示例：

```
try
{
  throw new AppError("Encountered Custom AppError", 29);
}
catch (error:AppError)
{
  trace(error.errorID + ": " + error.message)
}
```

注意

如果要在子类中覆盖 `Error.toString()` 方法，则需要在该方法中使用 ... 参数 （表示还有其它参数）。 ECMAScript (ECMA-262) 第 3 版语言规范按此方式定义了 `Error.toString()` 方法，而为了与该规范向后兼容， ActionScript 3.0 也采用了相同的定义方式。因此，在覆盖 `Error.toString()` 方法时，必须精确匹配各个参数。运行时无需向 `toString()` 方法传递任何参数，因为传递的参数都会被忽略。

# 响应错误事件和状态

在 ActionScript 3.0 中，对错误处理最为明显的一项改进就是支持对异步运行时错误予以响应。（有关异步错误的定义，请参阅第 218 页的 “错误类型”。）

可以创建事件侦听器和事件处理函数来响应错误事件。对于许多类来说，它们调度错误事件的方式与调度其它事件的方式相同。例如，一般情况下，XMLSocket 类实例调度三种类型的事件：`Event.CLOSE`、`Event.CONNECT` 和 `DataEvent.DATA`。但是，当发生问题时，XMLSocket 类还可以调度 `IOErrorEvent.IOError` 或 `SecurityErrorEvent.SECURITY_ERROR`。有关事件侦听器和事件处理函数的详细信息，请参阅第 267 页的第 10 章 “处理事件”。

错误事件分为两类：

- 扩展 ErrorEvent 类的错误事件

  flash.events.ErrorEvent 类包含用于管理有关网络和通信操作的 Flash Player 运行时错误的属性和方法。AsyncErrorEvent、IOErrorEvent 和 SecurityErrorEvent 类扩展了 ErrorEvent 类。如果使用的是 Flash Player 的调试版，则会出现一个对话框，向您告知播放器在运行时遇到的、没有侦听器函数的错误事件。

- 基于状态的错误事件

  基于状态的错误事件与网络和通信类的 `netStatus` 和 `status` 属性有关。如果 Flash Player 在读写数据时遇到问题，`netStatus.info.level` 或 `status.level` 属性（取决于使用的类对象）的值将被设置为值 `"error"`。可以通过检查事件处理函数中的 `level` 属性是否包含值 `"error"` 来响应此错误。

## 处理错误事件

ErrorEvent 类及其子类包含用于处理 Flash Player 尝试读写数据时调度的错误的错误类型。

下例同时使用了 `try..catch` 语句和错误事件处理函数来显示试图读取本地文件时检测到的错误。您可以在由 “在此处添加您的错误处理代码” 注释所指示的位置处添加更复杂的处理代码，以便为用户提供处理选项或者自动处理错误：

```
package
{
    import flash.display.Sprite;
    import flash.errors.IOError;
    import flash.events.IOErrorEvent;
    import flash.events.TextEvent;
    import flash.media.Sound;
    import flash.media.SoundChannel;
    import flash.net.URLRequest;
    import flash.text.TextField;

    public class LinkEventExample extends Sprite
    {
```

```
    private var myMP3:Sound;
    public function LinkEventExample()
    {
      myMP3 = new Sound();
      var list:TextField = new TextField();
      list.autoSize = TextFieldAutoSize.LEFT;
      list.multiline = true;
      list.htmlText = "<a href=\"event:track1.mp3\">Track 1</a><br>";
      list.htmlText += "<a href=\"event:track2.mp3\">Track 2</a><br>";
      addEventListener(TextEvent.LINK, linkHandler);
      addChild(list);
    }

    private function playMP3(mp3:String):void
    {
      try
      {
        myMP3.load(new URLRequest(mp3));
        myMP3.play();
      }
      catch (err:Error)
      {
        trace(err.message);
        // 在此处添加错误处理代码
      }
      myMP3.addEventListener(IOErrorEvent.IO_ERROR, errorHandler);
    }

    private function linkHandler(linkEvent:TextEvent):void
    {
      playMP3(linkEvent.text);
      // 在此处添加错误处理代码
    }

    private function errorHandler(errorEvent:IOErrorEvent):void
    {
      trace(errorEvent.text);
      // 在此处添加错误处理代码
    }
  }
}
```

## 处理状态更改事件

对于支持 `level` 属性的类， Flash Player 会动态更改 `netStatus.info.level` 或 `status.level` 属性的值。具有 `netStatus.info.level` 属性的类有 NetConnection、NetStream 和 SharedObject。具有 `status.level` 属性的类有 HTTPStatusEvent、Camera、 Microphone 和 LocalConnection。可以编写一个处理函数来响应 `level` 值的更改并跟踪通信错误。

以下示例使用 `netStatusHandler()` 函数测试 `level` 属性的值。如果 `level` 属性指示遇到错误，该代码将跟踪消息 “Video stream failed”（视频流失败）。

```
package
{
  import flash.display.Sprite;
  import flash.events.NetStatusEvent;
  import flash.events.SecurityErrorEvent;
  import flash.media.Video;
  import flash.net.NetConnection;
  import flash.net.NetStream;

  public class VideoExample extends Sprite
  {
    private var videoUrl:String = "Video.flv";
    private var connection:NetConnection;
    private var stream:NetStream;

    public function VideoExample()
    {
      connection = new NetConnection();
      connection.addEventListener(NetStatusEvent.NET_STATUS,
  netStatusHandler);
      connection.addEventListener(SecurityErrorEvent.SECURITY_ERROR,
  securityErrorHandler);
      connection.connect(null);
    }

    private function netStatusHandler(event:NetStatusEvent):void
    {
      if (event.info.level = "error")
      {
        trace(" 视频流失败 ")
      }
      else
      {
        connectStream();
      }
    }

    private function securityErrorHandler(event:SecurityErrorEvent):void
```

```
    {
      trace("securityErrorHandler: " + event);
    }

    private function connectStream():void
    {
      var stream:NetStream = new NetStream(connection);
      var video:Video = new Video();
      video.attachNetStream(stream);
      stream.play(videoUrl);
      addChild(video);
    }
  }
}
```

# 比较错误类

ActionScript 提供了许多预定义的错误类。其中的许多错误类由 Flash Player 使用，但您在自己的代码中也可以使用这些错误类。在 ActionScript 3.0 中，有两种主要类型的错误类：ActionScript 核心错误类和 flash.error 包错误类。核心错误类由 ECMAScript (ECMA-262) 第 4 版语言规范草案定义。flash.error 包包含为有助于 ActionScript 3.0 应用程序开发和调试工作而引入的附加类。

## ECMAScript 核心错误类

ECMAScript 核心错误类包括 Error、 EvalError、 RangeError、 ReferenceError、 SyntaxError、 TypeError 以及 URIError 类。其中的每个类均位于顶级命名空间中。

| 类名称 | 说明 | 备注 |
|---|---|---|
| Error | Error 类可用于引发异常，它是 ECMAScript 中所定义的其它异常类 （EvalError、 RangeError、 ReferenceError、 SyntaxError、 TypeError 和 URIError）的基类。 | Error 类用作 Flash Player 所引发的所有运行时错误的基类，并建议将其用作任何自定义错误类的基类。 |
| EvalError | 如果为 Function 类的构造函数传递了任何参数，或者用户代码调用 `eval()` 函数，则会引发 EvalError 异常。 | 在 ActionScript 3.0 中，取消了对 `eval()` 函数的支持，因此，如果试图使用该函数，则会引发错误。<br>Flash Player 的先前版本使用 `eval()` 函数来按照名称访问变量、属性、对象或影片剪辑。 |
| RangeError | 如果数值不在可接受的范围内，则会引发 RangeError 异常。 | 例如，当延迟是负数或不是有限值时，Timer 类就会引发 RangeError。试图在无效深度处添加显示对象也会引发 RangeError。 |

| 类名称 | 说明 | 备注 |
|---|---|---|
| ReferenceError | 如果试图对密封（非动态）对象引用未定义的属性，则会引发 ReferenceError 异常。试图访问未定义的属性时，ActionScript 3.0 之前的 ActionScript 编译器版本并不会引发错误。但鉴于新的 ECMAScript 规范规定在这种情况下应引发错误，因此 ActionScript 3.0 将引发 ReferenceError 异常。 | 由于访问未定义变量而引发异常表明存在潜在的错误，有助于提高软件质量。不过，如果您不习惯于对变量进行初始化，则对于 ActionScript 这种新的行为，您可能需要改变一些代码编写习惯。 |
| SyntaxError | 如果 ActionScript 代码发生解析错误，则会引发 SyntaxError 异常。<br>有关详细信息，请参阅位于以下地址的 ECMAScript (ECMA-262) 第 3 版语言规范（在第 4 版推出之前）的 15.11.6.4 节：www.ecma-international.org/publications/standards/Ecma-262.htm，以及位于以下地址的 ECMAScript for XML (E4X) 规范（ECMA-357 第 2 版）的 10.3.1 节：www.ecma-international.org/publications/standards/Ecma-357.htm。 | 在以下情况下会引发 SyntaxError：<br>• 当 RegExp 类解析的是无效正则表达式时，ActionScript 会引发 SyntaxError 异常。<br>• 当 XMLDocument 类解析的是无效 XML 时， ActionScript 会引发 SyntaxError 异常。 |
| TypeError | 如果操作数的实际类型与所需类型不同，则会引发 TypeError 异常。<br>有关详细信息，请参阅位于以下地址的 ECMAScript 规范的 15.11.6.5 节：www.ecma-international.org/publications/standards/Ecma-262.htm，以及位于以下地址的 E4X 规范的 10.3 节：www.ecma-international.org/publications/standards/Ecma-357.htm。 | 在以下情况下会引发 TypeError：<br>• 无法将函数的实际参数或方法强制为正式参数类型。<br>• 值已赋给变量，但无法强制为变量的类型。<br>• `is` 或 `instanceof` 运算符的右侧不是有效类型。<br>• 非法使用了 `super` 关键字。<br>• 属性查找生成了多个绑定，因此造成该查找不明确。<br>• 对不兼容的对象调用了某种方法。例如，如果 RegExp 类中的某个方法被 “转接” 到通用对象上，然后调用该方法，则将引发 TypeError 异常。 |
| URIError | 如果采用与某个全局 URI 处理函数的定义相矛盾的方式使用该函数，则会引发 URIError 异常。<br>有关详细信息，请参阅位于以下地址的 ECMAScript 规范的 15.11.6.6 节：www.ecma-international.org/publications/standards/Ecma-262.htm。 | 在以下情况下会引发 URIError：<br>为需要有效 URI 的 Flash Player API 函数（如 `Socket.connect()`）指定了无效的 URI。 |

## ActionScript 核心错误类

除 ECMAScript 核心错误类之外，ActionScript 还增加了几个自己的类，用于 ActionScript 特定的错误条件和错误处理功能。

由于这些类是对 ECMAScript 第 4 版语言规范草案的 ActionScript 语言扩展，并且可能会被收入该规范草案中，因此将这些类保留在顶级，而不是放置在某个包（如 flash.error）中。

| 类名称 | 描述 | 注释 |
| --- | --- | --- |
| ArgumentError | ArgumentError 类表示在函数调用期间提供的参数与为该函数定义的参数不一致时发生的错误。 | 以下是一些参数错误示例：<br>• 向方法提供的参数过少或过多。<br>• 参数应是某个枚举值，但实际上不是。 |
| SecurityError | 如果发生安全违规且访问被拒绝时，则会引发 SecurityError 异常。 | 以下是一些安全错误示例：<br>• 通过安全沙箱边界进行未经授权的属性访问或方法调用。<br>• 尝试访问安全沙箱不允许的 URL。<br>• 在不存在策略文件的情况下尝试与未经授权的端口号（例如低于 1024 的端口）进行套接字连接。<br>• 尝试访问用户的摄像头或麦克风，而访问此类设备的请求被用户拒绝。 |
| VerifyError | 如果遇到格式不正确或损坏的 SWF 文件，则会引发 VerifyError 异常。 | 当某个 SWF 文件加载另一个 SWF 文件时，父 SWF 可能会捕获由加载的 SWF 生成的 VerifyError。 |

## flash.error 包的错误类

flash.error 包中包含的错误类是 Flash Player API 的一部分。与刚刚描述的错误类不同，flash.error 包可以与特定于 Flash Player 的错误事件进行通信。

| 类名称 | 描述 | 注释 |
| --- | --- | --- |
| EOFError | 如果尝试读取的内容超出可用数据的末尾，则会引发 EOFError 异常。 | 例如，当调用 IDataInput 接口中的一个读取方法，而数据不足以满足读取请求时，将引发 EOFError。 |
| IllegalOperationError | 如果方法未实现或者实现中未涵盖当前用法，则会引发 IllegalOperationError 异常。 | 以下是非法操作错误异常的示例：<br>• 基类（如 DisplayObjectContainer）提供的功能比舞台可以支持的功能多。例如，如果试图在舞台上获取或设置一个蒙版（使用 `stage.mask`），Flash Player 将引发 IllegalOperationError，并显示消息"Stage 类未实现此属性或方法"。<br>• 子类继承了不需要且不想支持的方法。<br>• 在没有辅助功能支持的情况下编译 Flash Player 之后，又调用了某些辅助功能方法。<br>• 从 Flash Player 的运行时版本中调用仅用于创作的功能。<br>• 试图为放在时间轴上的对象设置名称。 |
| IOError | 如果发生某种类型的 I/O 异常，则会引发 IOError 异常。 | 例如，当试图对尚未连接或已断开连接的套接字进行读 / 写操作时，将引发此错误。 |
| MemoryError | 如果内存分配请求失败，则会引发 MemoryError 异常。 | 默认情况下， ActionScript Virtual Machine 2 不会对 ActionScript 程序可以分配的内存量强加限制。对于桌面 PC，内存分配失败的情况并不常见。当系统无法分配操作所需的内存时，将会看到这样的错误。因此，对于桌面 PC，这种异常很少见，除非分配请求非常大，如分配 3 GB 字节的请求便无法实现，因为 32 位 Microsoft® Windows® 应用程序只能访问 2 GB 的地址空间。 |

| 类名称 | 描述 | 注释 |
| --- | --- | --- |
| ScriptTimeoutError | 如果达到了 15 秒的脚本超时间隔，则会引发 ScriptTimeoutError 异常。通过捕获 ScriptTimeoutError 异常，可以更加妥善地处理脚本超时。如果没有异常处理函数，未捕获异常的处理函数将显示一个带有错误消息的对话框。 | 为防止恶意开发者捕获这种异常并导致无限循环，仅能够捕获特定脚本运行过程中引发的第一个 ScriptTimeoutError 异常。后面的 ScriptTimeoutError 异常将无法被开发者的代码捕获，并会立即转到未捕获异常的处理函数。 |
| StackOverflowError | 如果脚本可用堆栈已经用尽，则会引发 StackOverflowError 异常。 | StackOverflowError 异常可能表明发生了无限递归。 |

# 示例：CustomErrors 应用程序

CustomErrors 应用程序展示了构建应用程序时处理自定义错误的一些技巧。这些方法包括：

- 验证 XML 包
- 编写自定义错误
- 引发自定义错误
- 引发错误时通知用户

要获取该范例的应用程序文件，请访问 www.adobe.com/go/learn_programmingAS3samples_flash_cn。可以在 Samples/CustomError 文件夹中找到 CustomErrors 应用程序的文件。该应用程序包含以下文件：

| 文件 | 描述 |
| --- | --- |
| CustomErrors.mxml<br>或<br>CustomErrors.fla | Flash (FLA) 或 Flex (MXML) 中的主应用程序文件。 |
| com/example/programmingas3/errors/ApplicationError.as | 一个类，用作 FatalError 类和 WarningError 类的基错误类。 |
| com/example/programmingas3/errors/FatalError.as | 一个类，定义该应用程序可以引发的 FatalError 错误。该类扩展了自定义的 ApplicationError 类。 |
| com/example/programmingas3/errors/Validator.as | 一个类，定义了用于验证用户提供的员工 XML 包的单个方法。 |
| com/example/programmingas3/errors/WarningError.as | 一个类，定义了该应用程序可以引发的 WarningError 错误。该类扩展了自定义的 ApplicationError 类。 |

## CustomErrors 应用程序概述

CustomErrors.mxml 文件包含该自定义错误应用程序的用户界面和一些逻辑。一旦调度了该应用程序的 `creationComplete` 事件，便会调用 `initApp()` 方法。该方法定义将由 Validator 类验证的示例 XML 包。以下显示的是 `initApp()` 方法的代码：

```
private function initApp():void
{
    employeeXML =
        <employee id="12345">
            <firstName>John</firstName>
            <lastName>Doe</lastName>
            <costCenter>12345</costCenter>
            <costCenter>67890</costCenter>
        </employee>;
}
```

稍后，将在舞台上的 TextArea 组件实例中显示该 XML 包。这样就可以在尝试重新验证该 XML 包之前对其进行修改。

用户单击 Validate 按钮时，将调用 `validateData()` 方法。该方法使用 Validator 类中的 `validateEmployeeXML()` 方法来验证员工 XML 包。以下显示的是 `validateData()` 方法的代码：

```
public function validateData():void
{
    try
    {
        var tempXML:XML = XML(xmlText.text);
        Validator.validateEmployeeXML(tempXML);
        status.text = "The XML was successfully validated.";
    }
    catch (error:FatalError)
    {
        showFatalError(error);
    }
    catch (error:WarningError)
    {
        showWarningError(error);
    }
    catch (error:Error)
    {
        showGenericError(error);
    }
}
```

首先，使用 TextArea 组件实例 xmlText 内容创建一个临时的 XML 对象。接下来，调用自定义 Validator 类 (com.example.programmingas3/errors/Validator.as) 中的 validateEmployeeXML() 方法，并将这个临时 XML 对象作为参数传递给该方法。如果这个 XML 包是有效的， Label 组件实例 status 便会显示一条成功消息，然后应用程序退出。如果 validateEmployeeXML() 方法引发了一个自定义错误 （即发生 FatalError、WarningError 或一般的 Error），则会执行相应的 catch 语句并调用 showFatalError()、showWarningError() 或 showGenericError() 方法。这几种方法均会在一个 Alert 组件中显示相应的消息，以通知用户发生了特定错误。每个方法还会用具体的消息更新 Label 组件实例 status。

如以下代码所示，如果尝试验证员工 XML 包时发生致命错误，则会在一个 Alert 组件中显示错误消息，并且禁用 TextArea 组件实例 xmlText 和 Button 组件实例 validateBtn:

```
public function showFatalError(error:FatalError):void
{
    var message:String = error.message + "\n\n" + "Click OK to end.";
    var title:String = error.getTitle();
    Alert.show(message, title);
    status.text = "This application has ended.";
    this.xmlText.enabled = false;
    this.validateBtn.enabled = false;
}
```

如果发生的是警告错误而不是致命错误，则会在 Alert 组件实例中显示错误消息，但不会禁用 TextField 和 Button 组件实例。showWarningError() 方法会在 Alert 组件实例中显示自定义的错误消息。该消息还会让用户决定是希望继续验证 XML 还是终止脚本。以下显示的是 showWarningError() 方法的代码:

```
public function showWarningError(error:WarningError):void
{
    var message:String = error.message + "\n\n" + "Do you want to exit this
    application?";
    var title:String = error.getTitle();
    Alert.show(message, title, Alert.YES | Alert.NO, null, closeHandler);
    status.text = message;
}
```

用户使用 Yes 或 No 按钮关闭 Alert 组件实例时，将调用 closeHandler() 方法。以下显示的是 closeHandler() 方法的代码:

```
private function closeHandler(event:CloseEvent):void
{
    switch (event.detail)
    {
        case Alert.YES:
            showFatalError(new FatalError(9999));
            break;
        case Alert.NO:
            break;
    }
}
```

如果用户选择单击警告错误 Alert 对话框中的 Yes 按钮来终止脚本，则会引发 FatalError，从而导致应用程序终止运行。

## 构建自定义验证程序

自定义的 Validator 类仅包含一个方法：validateEmployeeXML()。validateEmployeeXML() 方法使用单一参数 employee，该参数为要验证的 XML 包。validateEmployeeXML() 方法的代码如下所示：

```
public static function validateEmployeeXML(employee:XML):void
{
  // checks for the integrity of items in the XML
  if (employee.costCenter.length() < 1)
  {
    throw new FatalError(9000);
  }
  if (employee.costCenter.length() > 1)
  {
    throw new WarningError(9001);
  }
  if (employee.ssn.length() != 1)
  {
    throw new FatalError(9002);
  }
}
```

员工必须属于一个（且只能属于一个）成本中心才能通过验证。如果员工不属于任何成本中心，该方法将引发 FatalError，该异常将向上冒泡到应用程序主文件中的 validateData() 方法。如果员工属于多个成本中心，则引发 WarningError。该 XML 验证程序最后检查用户是否只定义了一个社会安全号码（XML 包中的 ssn 节点）。如果具有不止一个 ssn 节点，则引发 FatalError 错误。

您可以向 validateEmployeeXML() 方法添加其它检查，例如，确保 ssn 节点包含的是有效号码，或者员工至少定义了一个电话号码和一个电子邮件地址，并且这两个值都是有效的。您还可以对该 XML 进行修改，使每个员工具有唯一的 ID 并指定其管理者的 ID。

## 定义 ApplicationError 类

ApplicationError 类用作 FatalError 类和 WarningError 类的基类。ApplicationError 类扩展了 Error 类，并且定义了自己的自定义方法和属性，其中包括定义一个错误 ID、严重程度以及包含自定义错误代码和消息的 XML 对象。该类还定义了两个静态常量，它们用于定义每种错误类型的严重程度。

ApplicationError 类的构造函数方法如下所示：

```
public function ApplicationError()
{
    messages =
        <errors>
            <error code="9000">
                <![CDATA[Employee must be assigned to a cost center.]]>
            </error>
            <error code="9001">
                <![CDATA[Employee must be assigned to only one cost center.]]>
            </error>
            <error code="9002">
                <![CDATA[Employee must have one and only one SSN.]]>
            </error>
            <error code="9999">
                <![CDATA[The application has been stopped.]]>
            </error>
        </errors>;
}
```

XML 对象中的每个错误节点都包含一个唯一的数值代码和一条错误消息。使用 E4X 可以很容易地通过错误代码查找到相应的错误消息，如以下 `getMessageText()` 方法所示：

```
public function getMessageText(id:int):String
{
    var message:XMLList = messages.error.(@code == id);
    return message[0].text();
}
```

`getMessageText()` 方法仅使用一个整数参数 `id` 并返回一个字符串。该 `id` 参数就是要查找的错误的错误代码。例如，传递等于 9001 的 `id` 值将得到错误消息“Employee must be assigned to only one cost center”（只能为员工指定一个成本中心）。如果多个错误具有同一错误代码，ActionScript 将只为找到的第一个结果返回错误消息（所返回 XMLList 对象中的 `message[0]`）。

该类中的下一个方法 `getTitle()` 不使用任何参数，并返回一个字符串值，其中包含此特定错误的错误 ID。该值用于 Alert 组件的标题中，让您轻松识别在验证 XML 包期间发生的具体错误。以下显示的是 `getTitle()` 方法的代码：

```
public function getTitle():String
{
    return "Error #" + id;
}
```

ApplicationError 类中的最后一个方法是 `toString()`。该方法覆盖 Error 类中定义的函数，以便您可以自定义错误消息的显示。该方法将返回一个字符串，用于识别所发生错误的具体编号和消息。

```
public override function toString():String
{
  return "[APPLICATION ERROR #" + id + "] " + message;
}
```

## 定义 FatalError 类

FatalError 类扩展了自定义的 ApplicationError 类并定义三个方法：FatalError 构造函数、`getTitle()` 和 `toString()`。第一个方法（即 FatalError 构造函数）仅使用一个整数参数 `errorID`，并使用 ApplicationError 类中定义的静态常量设置错误的严重程度，另外还调用 ApplicationError 类中的 `getMessageText()` 方法获取特定错误的错误消息。FatalError 构造函数如下所示：

```
public function FatalError(errorID:int)
{
  id = errorID;
  severity = ApplicationError.FATAL;
  message = getMessageText(errorID);
}
```

FatalError 类中的下一个方法 `getTitle()` 覆盖先前在 ApplicationError 类中定义的 `getTitle()` 方法，并在标题后面追加文字 "-- FATAL"以通知用户发生了致命错误。`getTitle()` 方法如下所示：

```
public override function getTitle():String
{
  return "Error #" + id + " -- FATAL";
}
```

该类中的最后一个方法 `toString()` 覆盖 ApplicationError 类中定义的 `toString()` 方法。`toString()` 方法如下所示：

```
public override function toString():String
{
  return "[FATAL ERROR #" + id + "] " + message;
}
```

## 定义 WarningError 类

WarningError 类扩展了 ApplicationError 类，它与 FatalError 类几乎完全相同，只是字符串稍微有些不同，另外它还将错误严重程度设置为 ApplicationError.WARNING 而不是 ApplicationError.FATAL，如以下代码所示：

```
public function WarningError(errorID:int)
{
  id = errorID;
  severity = ApplicationError.WARNING;
  message = super.getMessageText(errorID);
}
```

第 9 章

# 使用正则表达式

9

正则表达式描述用于查找和处理字符串中的匹配文本的模式。正则表达式类似于字符串，但是可以包含特殊代码以描述模式和重复。例如，下面的正则表达式与以字符 A 开头并且后跟一个或多个连续数字的字符串匹配：

```
/A\d+/
```

本章介绍了构造正则表达式的基本语法。但是实际上，正则表达式可能非常复杂且具有许多细微差别。您可以从网上或者书店中找到有关正则表达式的详细资料。切记，不同的编程环境实现正则表达式的方式也不同。ActionScript 3.0 按照 ECMAScript 第 3 版语言规范 (ECMA-262) 中的定义实现正则表达式。

## 目录

# 正则表达式基础知识

## 使用正则表达式简介

正则表达式描述字符模式。通常，正则表达式用于验证文本值是否符合特定模式（例如，验证用户输入的电话号码位数是否正确），或者替换与特定模式匹配的部分文本值。

正则表达式可能非常简单。例如，假设您要确认特定字符串与 “ABC” 是否匹配，或者要使用某些其它文本替换字符串中出现的每个 “ABC”。在这种情况下，您可以使用以下正则表达式，它定义了依次包含字母 A、 B 和 C 的模式：

```
/ABC/
```

请注意，正则表达式文本是使用正斜杠 (/) 界定的。

正则表达式模式也可能非常复杂，有时候表面上看起来晦涩难懂，例如，以下与有效电子邮件地址匹配的表达式：

```
/([0-9a-zA-Z]+[-._+&])*[0-9a-zA-Z]+@([-0-9a-zA-Z]+[.])+[a-zA-Z]{2,6}/
```

通常，您使用正则表达式在字符串中搜索模式以及替换字符。在这些情况下，您将创建一个正则表达式对象，并将其作为几个 String 类方法之一的参数。下列 String 类方法将正则表达式作为参数：`match()`、 `replace()`、 `search()` 和 `split()`。有关这些方法的详细信息，请参阅第 179 页的 “在字符串中查找模式并替换子字符串”。

RegExp 类包含以下方法：`test()` 和 `exec()`。有关详细信息，请参阅第 260 页的 “对字符串使用正则表达式的方法”。

## 常见正则表达式任务

正则表达式具有几种常见用途，本章详细介绍了这些用途：

- 创建正则表达式模式
- 在模式中使用特殊字符
- 标识多个字符的序列（如 “两位数” 或 “介于 7 到 10 个字母之间”）
- 标识字母或数字范围中的任何字母（如 “从 *a* 到 *m* 的任何字母”）
- 标识可能的字符集中的字符
- 标识子序列（模式内的片断）
- 基于模式匹配和替换文件

## 重要概念和术语

以下参考列表包含将会在本章中使用的重要术语：

- 转义字符 (Escape character)：此字符指示应将后面的字符视为元字符，而不是字面字符。在正则表达式语法中，反斜杠字符 (\) 就是转义字符，因此反斜杠后跟另一个字符是一个特殊代码，而不仅仅是字符本身。
- 标志 (Flag)：此字符指定有关应如何使用正则表达式模式的一些选项，如是否区分大写和小写字符。
- 元字符 (Metacharacter)：在正则表达式模式中具有特殊含义的字符，它与从字面意义上在模式中表示该字符相对。
- 数量表示符 (Quantifier)：一个或几个字符，指示应将模式部分重复多少次。例如，使用数量表示符来指定美国邮政编码应包含 5 个或 9 个数字。
- 正则表达式 (Regular expression)：用于定义字符模式的程序语句，它可用来确认其它字符串是否与该模式匹配，或者替换字符串的一部分。

## 完成本章中的示例

学习本章的过程中，您可能想要自己动手测试一些示例代码清单。由于本章中的代码清单主要由正则表达式模式组成，因此测试示例涉及以下几个步骤：

1. 创建一个新的 Flash 文档。
2. 选择一个关键帧并打开 “动作” 面板。
3. 创建一个 RegExp （正则表达式）变量，例如：
   ```
   var pattern:RegExp = /ABC/;
   ```
4. 复制示例的模式并将其作为 RegExp 变量的值进行分配。例如，在上一行代码中，模式是等号右侧的代码的一部分，不包含分号 (`/ABC/`)。
5. 创建一个或多个包含适合于测试正则表达式的字符串的 String 变量。例如，如果要创建正则表达式来测试有效的电子邮件地址，则创建几个包含有效和无效电子邮件地址的 String 变量：
   ```
   var goodEmail:String = "bob@example.com";
   var badEmail:String = "5@$2.99";
   ```
6. 添加几行测试 String 变量的代码，以确定它们是否与正则表达式模式匹配。这些将是您希望通过使用 `trace()` 函数或通过写入舞台的文本字段而输出到屏幕上的值。
   ```
   trace(goodEmail, " is valid:", pattern.test(goodEmail));
   trace(badEmail, " is valid:", pattern.test(badEmail));
   ```

例如，假设 pattern 为有效的电子邮件地址定义了正则表达式模式，前面几行代码将此文本写入 “输出” 面板：

```
bob@example.com is valid: true
5@$2.99 is valid: false
```

有关通过将值写入舞台上的文本字段实例或使用 trace() 函数将值输出到 “输出” 面板以测试这些值的详细信息，请参阅第 53 页的 “测试本章内的示例代码清单”。

# 正则表达式语法

本节介绍了 ActionScript 正则表达式语法的全部元素。正如您所看到的一样，正则表达式可能非常复杂且具有许多细微差别。您可以从网上或者书店中找到有关正则表达式的详细资料。切记，不同的编程环境实现正则表达式的方式也不同。ActionScript 3.0 按照 ECMAScript 第 3 版语言规范 (ECMA-262) 中的定义实现正则表达式。

通常，您要使用的正则表达式是与比较复杂的模式匹配，而不是与简单的字符串匹配。例如，下面的正则表达式定义了由字母 A、 B 和 C 依次排列且后跟数字的模式：

```
/ABC\d/
```

\d 代码表示“任意数字”。反斜杠 (\) 字符称为转义字符，它与后面的字符（在本例中为字母 d）配合使用，在正则表达式中具有特殊含义。本章将介绍这些转义字符序列以及其它一些正则表达式语法特征。

下面的正则表达式定义了由字母 ABC 后跟任意数目的数字组成的模式 （注意星号）：

```
/ABC\d*/
```

星号字符 (*) 是 “元字符”。元字符是在正则表达式中具有特殊含义的字符。星号是一种称为 “数量表示符” 的特殊类型元字符，用于定义某个字符或一组字符重复的次数。有关详细信息，请参阅第 252 页的 “数量表示符”。

除了它的模式外，正则表达式还可以包含标志，用于指定正则表达式的匹配方式。例如，下面的正则表达式使用 i 标志指定正则表达式在匹配字符串中忽略大小写：

```
/ABC\d*/i
```

有关详细信息，请参阅第 257 页的 “标志和属性”。

您可以通过以下 String 类方法使用正则表达式：match()、 replace() 和 search()。有关这些方法的详细信息，请参阅第 179 页的 “在字符串中查找模式并替换子字符串”。

## 创建正则表达式实例

有两种方法可以创建正则表达式实例。一种方法是使用正斜杠字符 (/) 来界定正则表达式，另一种是使用 new 构造函数。例如，以下两个正则表达式是等效的：

```
var pattern1:RegExp = /bob/i;
var pattern2:RegExp = new RegExp("bob", "i");
```

正斜杠界定正则表达式的方式与用引号界定字符串文本的方式相同。正斜杠内的正则表达式部分定义"模式"。正则表达式还可以在后一个界定斜杠后包含"标志"。这些标志也看作是正则表达式的一部分，但是它们独立于模式。

使用 new 构造函数时，使用两个字符串来定义正则表达式。第一个字符串定义模式，第二个字符串定义标志，如下例所示：

```
var pattern2:RegExp = new RegExp("bob", "i");
```

如果在使用正斜杠界定符定义的正则表达式中包含正斜杠，则必须在正斜杠前面加上反斜杠 (\) 转义字符。例如，下面的正则表达式与模式 1/2 匹配：

```
var pattern:RegExp = /1\/2/;
```

要在使用 new 构造函数定义的正则表达式中包含引号，您必须在引号前面加上反斜杠 (\) 转义字符（就像定义任何 String 文本一样）。例如，下面的正则表达式与模式 eat at "joe's" 匹配：

```
var pattern1:RegExp = new RegExp("eat at \"joe's\"", "");
var pattern2:RegExp = new RegExp('eat at "joe\'s"', "");
```

请勿在使用正斜杠界定符定义的正则表达式中对引号使用反斜杠转义字符。同样地，不要在使用 new 构造函数定义的正则表达式中对正斜杠使用转义字符。下列正则表达式是等效的，它们都定义了模式 1/2 "joe's"：

```
var pattern1:RegExp = /1\/2 "joe's"/;
var pattern2:RegExp = new RegExp("1/2 \"joe's\"", "");
var pattern3:RegExp = new RegExp('1/2 "joe\'s"', '');
```

此外，在使用 new 构造函数定义的正则表达式中，要使用以反斜杠 (\) 字符开头的元序列（例如，\d，用于匹配任何数字），请键入两个反斜杠字符：

```
var pattern:RegExp = new RegExp("\\d+", ""); // 匹配一个或多个数字
```

在本实例中您必须键入两个反斜杠字符，因为 RegExp() 构造函数方法的第一个参数是字符串，而在字符串文本中必须键入两个反斜杠字符才能将其识别为单个反斜杠字符。

后面几节将介绍定义正则表达式模式的语法。

有关标志的详细信息，请参阅第 257 页的"标志和属性"。

## 字符、元字符和元序列

最简单的正则表达式是与字符序列匹配的表达式，如以下示例中所示：

```
var pattern:RegExp = /hello/;
```

但是，下列字符 （称为元字符）在正则表达式中具有特殊含义：

```
^ $ \ . * + ? ( ) [ ] { } |
```

例如，下面的正则表达式所匹配的是字母 A 后跟字母 B 的零个或多个实例 （星号元字符表示重复）再跟字母 C：

```
/AB*C/
```

要在正则表达式模式中包含元字符以使其不具有特殊含义，您必须使用反斜杠 (\) 转义字符。例如，下面的正则表达式与顺序依次为字母 A、字母 B、星号和字母 C 的模式匹配：

```
var pattern:RegExp = /AB\*C/;
```

"元序列"与元字符类似，在正则表达式中具有特殊含义。元序列由多个字符组成。以下几节提供了有关使用元字符和元序列的详细信息。

### 关于元字符

下表总结了可以在正则表达式中使用的元字符：

| 元字符 | 描述 |
|---|---|
| ^ （尖号） | 匹配字符串的开头。设置 m (multiline) 标志后，尖号还匹配行的开头 （请参阅第 258 页的 "m (multiline) 标志"）。请注意，尖号用在字符类的开头时表示符号反转而非字符串的开头。有关详细信息，请参阅第 250 页的 "字符类"。 |
| $ （美元符号） | 匹配字符串的结尾。设置 m (multiline) 标志后， $ 还匹配换行 (\n) 字符前面的位置。有关详细信息，请参阅第 258 页的 "m (multiline) 标志"。 |
| \ （反斜杠） | 对特殊字符的特殊元字符含义进行转义。<br>此外，如果要在正则表达式文本中使用正斜杠字符，也要使用反斜杠字符，例如， /1\/2/ 匹配字符 1 后跟正斜杠字符和字符 2。 |
| . （点） | 匹配任意单个字符。<br>只有设置 s (dotall) 标志时，点才匹配换行字符 (\n)。有关详细信息，请参阅第 258 页的 "s (dotall) 标志"。 |
| * （星号） | 匹配前面重复零次或多次的项目。<br>有关详细信息，请参阅第 252 页的 "数量表示符"。 |
| + （加号） | 匹配前面重复一次或多次的项目。<br>有关详细信息，请参阅第 252 页的 "数量表示符"。 |
| ? （问号） | 匹配前面重复零次或一次的项目。<br>有关详细信息，请参阅第 252 页的 "数量表示符"。 |

| 元字符 | 描述 |
| --- | --- |
| ( 和 ) | 在正则表达式中定义组。以下情况下使用组：<br>• 限制逻辑“或”字符 \| 的范围：/(a\|b\|c)d/<br>• 定义数量表示符的范围：/(walla.){1,2}/<br>• 用在逆向引用中。例如，下面的正则表达式中的 \1 匹配模式的第一个括号组中的匹配内容：<br>/(\w*) is repeated: \1/<br>有关详细信息，请参阅第 254 页的“组”。 |
| [ 和 ] | 定义字符类，字符类定义单个字符可能的匹配：<br>/[aeiou]/ 匹配所指定字符中的任意一个。<br>在字符类中，使用连字符 (-) 指定字符的范围：<br>/[A-Z0-9]/ 匹配从 A 到 Z 的大写字母或 0 到 9 的数字。<br>在字符类中，插入反斜杠对 ] 和<br>- 字符进行转义：<br>/[+\-]\d+/ 匹配一个或多个数字前面的 + 或 -。<br>在字符类中，以下字符（通常为元字符）被看作一般字符（非元字符），不需要反斜杠：<br>/[$£]/ 匹配 $ 或 £。<br>有关详细信息，请参阅第 250 页的“字符类”。 |
| \| *（竖线）* | 用于逻辑“或”操作，匹配左侧或右侧的部分：<br>/abc\|xyz/ 匹配 abc 或 xyz。 |

## 关于元序列

元序列是在正则表达式模式中具有特殊含义的字符序列。下表说明了这些元序列：

| 元序列 | 描述 |
| --- | --- |
| {*n*}<br>{*n*,}<br>和<br>{*n*,*n*} | 指定前一项目的数值数量或数量范围：<br>/A{27}/ 匹配重复 27 次的字符 A。<br>/A{3,}/ 匹配重复 3 次或更多次的字符 A。<br>/A{3,5}/ 匹配重复 3 到 5 次的字符 A。<br>有关详细信息，请参阅第 252 页的“数量表示符”。 |
| \b | 匹配单词字符和非单词字符之间的位置。如果字符串中的第一个或最后一个字符是单词字符，则也匹配字符串的开头或结尾。 |
| \B | 匹配两个单词字符之间的位置。也匹配两个非单词字符之间的位置。 |
| \d | 匹配十进制数字。 |
| \D | 匹配除数字以外的任何字符。 |
| \f | 匹配换页符。 |
| \n | 匹配换行符。 |
| \r | 匹配回车符。 |

| 元序列 | 描述 |
|---|---|
| \s | 匹配任何空白字符（空格、制表符、换行符或回车符）。 |
| \S | 匹配除空白字符以外的任何字符。 |
| \t | 匹配制表符。 |
| \u*nnnn* | 匹配字符代码由十六进制数字 *nnnn* 指定的 Unicode 字符。例如，\u263a 是一个笑脸字符。 |
| \v | 匹配垂直换页符。 |
| \w | 匹配单词字符（A-Z、a-z、0-9 或 _）。请注意，\w 不匹配非英文字符，如 é、ñ 或 ç。 |
| \W | 匹配除单词字符以外的任何字符。 |
| \x*nn* | 匹配具有指定 ASCII 值（由十六进制数字 *nn* 定义）的字符。 |

# 字符类

可以使用字符类指定字符列表以匹配正则表达式中的一个位置。使用方括号（[ 和 ]）定义字符类。例如，下面的正则表达式定义了匹配 bag、beg、big、bog 或 bug 的字符类：

```
/b[aeiou]g/
```

## 字符类中的转义序列

通常在正则表达式中具有特殊含义的大多数元字符和元序列在字符类中"不具有"那些特殊含义。例如，在正则表达式中星号用于表示重复，但是出现在字符类中时则不具有此含义。下列字符类匹配星号本身以及列出的任何其它字符：

```
/[abc*123]/
```

但是，下表中列出的三个字符功能与元字符相同，在字符类中具有特殊含义：

| 元字符 | 在字符类中的含义 |
|---|---|
| ] | 定义字符类的结尾。 |
| - | 定义字符范围（请参阅第 251 页的 "字符类中字符的范围"）。 |
| \ | 定义元序列并撤销元字符的特殊含义。 |

对于要识别为字面字符（无特殊元字符含义）的任何字符，必须在该字符前面加反斜杠转义字符。例如，下面的正则表达式包含匹配四个符号（$、\、] 或 -）中任意一个符号的字符类。

```
/[$\\\]\-]/
```

除能够保持特殊含义的元字符外，下列元序列在字符类中也具有元序列功能：

| 元序列 | 在字符类中的含义 |
|---|---|
| \n | 匹配换行符。 |
| \r | 匹配回车符。 |
| \t | 匹配制表符。 |
| \u*nnnn* | 匹配具有指定 Unicode 代码点值（由十六进制数字 *nnnn* 定义）的字符。 |
| \x*nn* | 匹配具有指定 ASCII 值（由十六进制数字 *nn* 定义）的字符。 |

其它正则表达式元序列和元字符在字符类中看作普通字符。

## 字符类中字符的范围

使用连字符指定字符的范围，例如 A-Z、 a-z 或 0-9。这些字符必须在字符类中构成有效的范围。例如，下面的字符类匹配 a-z 范围内的任何一个字符或任何数字：

```
/[a-z0-9]/
```

您还可以使用 \x*nn* ASCII 字符代码通过 ASCII 值指定范围。例如，下面的字符类匹配扩展 ASCII 字符集中的任意字符（如 é 和 ê）：

```
/[\x80-\x9A]/
```

## 反转的字符类

如果在字符类的开头使用尖号 (^) 字符，则将反转该集合的意义，即未列出的任何字符都认为匹配。下面的字符类匹配除小写字母 (a-z) 或数字以外的任何字符：

```
/[^a-z0-9]/
```

必须在字符类的 “开头”键入尖号 (^) 字符以表示反转。否则，您只是将尖号字符添加到字符类的字符中。例如，下面的字符类匹配许多符号字符中的任意一个，其中包括尖号：

```
/[!.,#+*%$&^]/
```

## 数量表示符

使用数量表示符指定字符或序列在模式中的重复次数，如下所示：

| 数量表示符元字符 | 描述 |
|---|---|
| * （星号） | 匹配前面重复零次或多次的项目。 |
| + （加号） | 匹配前面重复一次或多次的项目。 |
| ? （问号） | 匹配前面重复零次或一次的项目。 |
| {*n*}<br>{*n*,}<br>和<br>{*n*,*n*} | 指定前一项目的数值数量或数量范围：<br>/A{27}/ 匹配重复 27 次的字符 A。<br>/A{3,}/ 匹配重复 3 次或更多次的字符 A。<br>/A{3,5}/ 匹配重复 3 到 5 次的字符 A。 |

您可以将数量表示符应用到单个字符、字符类或组：

- /a+/ 匹配重复一次或多次的字符 a。
- /\d+/ 匹配一个或多个数字。
- /[abc]+/ 匹配重复的一个或多个字符，这些字符可能是 a、 b 或 c 中的某个。
- /(very, )*/ 匹配重复零次或多次的后跟逗号和空格的单词 very。

您可以在应用数量表示符的括号组内使用数量表示符。例如，下面的数量表示符匹配诸如 word 和 word-word-word 的字符串：

```
/\w+(-\w+)*/
```

默认情况下，正则表达式执行所谓 “无限匹配” 。正则表达式中的任何子模式 （如 .*）都会尝试在字符串中匹配尽可能多的字符，然后再执行正则表达式的下一部分。例如，使用以下正则表达式和字符串：

```
var pattern:RegExp = /<p>.*<\/p>/;
str:String = "<p>Paragraph 1</p> <p>Paragraph 2</p>";
```

正则表达式匹配整个字符串：

```
<p>Paragraph 1</p> <p>Paragraph 2</p>
```

但是，假如您只想匹配一个 <p>...</p> 组，则可以通过以下操作实现：

```
<p>Paragraph 1</p>
```

在所有数量表示符后添加问号 (?) 以将其更改为所谓“惰性数量表示符”。例如，下面的正则表达式使用惰性数量表示符 *? 匹配 <p> 后跟数量最少 （惰性）的字符，再跟 </p> 的模式：

```
/<p>.*?<\/p>/
```

有关数量表示符，请牢记以下几点：

- 数量表示符 {0} 和 {0,0} 不会从匹配中排除项目。
- 不要结合使用多个数量表示符，例如 /abc+*/ 中。
- 除非设置 s (dotall) 标志，否则点 (.) 不能跨越多行，即使后跟 * 数量表示符。例如，请考虑使用以下代码：

```
var str:String = "<p>Test\n";
str += "Multiline</p>";
var re:RegExp = /<p>.*<\/p>/;
trace(str.match(re)); // null;

re = /<p>.*<\/p>/s;
trace(str.match(re));
  // 输出: <p>Test
  //          Multiline</p>
```

有关详细信息，请参阅第 258 页的 "s (dotall) 标志"。

## 逻辑 "或"

在正则表达式中使用 | （竖线）字符可使正则表达式引擎考虑其它匹配。例如，下面的正则表达式匹配单词 cat、dog、pig 和 rat 中的任意一个：

```
var pattern:RegExp = /cat|dog|pig|rat/;
```

您可以使用括号定义组以限制逻辑 "或" 字符 | 的范围。下面的正则表达式匹配 cat 后跟 nap 或 nip：

```
var pattern:RegExp = /cat(nap|nip)/;
```

有关详细信息，请参阅第 254 页的 "组"。

下面两个正则表达式是等效的，一个使用 | 逻辑 "或" 字符，另一个使用字符类（由 [ 和 ] 定义）：

```
/1|3|5|7|9/
/[13579]/
```

有关详细信息，请参阅第 250 页的 "字符类"。

## 组

您可以使用括号在正则表达式中指定组，如下所示：

```
/class-(\d*)/
```

组是模式的子部分。您可以使用组实现以下操作：

- 将数量表示符应用到多个字符。
- 界定要应用逻辑 “或”（通过使用 | 字符）的子模式。
- 捕获正则表达式中的子字符串匹配（例如，在正则表达式中使用 `\1` 以匹配先前匹配的组，或类似地在 String 类的 `replace()` 方法中使用 `$1`）。

下面几节将介绍有关这些组用法的详细信息。

## 使用带数量表示符的组

如果不使用组，数量表示符将应用到它前面的字符或字符类，如下所示：

```
var pattern:RegExp = /ab*/ ;
// 匹配字符 a 后跟
// 零个或多个字符 b

pattern = /a\d+/;
// 匹配字符 a 后跟
// 一个或多个数字

pattern = /a[123]{1,3}/;
// 匹配字符 a 后跟
// 一到三个 1、2 或 3
```

然而，您可以使用组将数量表示符应用到多个字符或字符类：

```
var pattern:RegExp = /(ab)*/;
// 匹配零个或多个字符 a
// 后跟字符 b，如 ababab

pattern = /(a\d)+/;
// 匹配一个或多个 a 后跟
// 数字，例如 a1a5a8a3

pattern = /(spam ){1,3}/;
// 匹配 1 到 3 个单词 spam 后跟空格
```

有关数量表示符的详细信息，请参阅第 252 页的 “数量表示符”。

## 使用带逻辑 “或”字符 (|) 的组

可以使用组来定义一组要应用逻辑 “或”字符 (|) 的字符，如下所示：

```
var pattern:RegExp = /cat|dog/;
// 匹配 cat 或 dog

pattern = /ca(t|d)og/;
// 匹配 catog 或 cadog
```

## 使用组捕获子字符串匹配

如果您在模式中定义标准括号组，则之后可以在正则表达式中引用它。这称为“逆向引用”，并且此类型的组称为 “捕获组”。例如，在下面的正则表达式中，序列 `\1` 匹配在捕获括号组中匹配的任意子字符串：

```
var pattern:RegExp = /(\d+)-by-\1/;
// 匹配字符串: 48-by-48
```

您可以通过键入 `\1`, `\2`, ... , `\99` 在正则表达式中指定最多 99 个此类逆向引用。

类似地，在 String 类的 `replace()` 方法中，可以使用 `$1-$99` 在替换字符串中插入捕获的组子字符串匹配：

```
var pattern:RegExp = /Hi, (\w+)\./;
var str:String = "Hi, Bob.";
trace(str.replace(pattern, "$1, hello."));
// 输出: Bob, hello.
```

此外，如果使用捕获组，RegExp 类的 `exec()` 方法和 String 类的 `match()` 方法将返回与捕获组匹配的子字符串：

```
var pattern:RegExp = /(\w+)@(\w+).(\w+)/;
var str:String = "bob@example.com";
trace(pattern.exec(str));
   // bob@test.com,bob,example,com
```

## 使用非捕获组和向前查找组

非捕获组是只用于分组的组，它不会被“收集”，也不会匹配有限的逆向引用。可以使用 `(?:` 和 `)` 来定义非捕获组，如下所示：

```
var pattern = /(?:com|org|net);
```

例如，注意在捕获组和非捕获组中加入 `(com|org)` 的区别 (`exec()` 方法在完全匹配后列出捕获组)：

```
var pattern:RegExp = /(\w+)@(\w+).(com|org)/;
var str:String = "bob@example.com";
trace(pattern.exec(str));
  // bob@test.com,bob,example,com
```

```
// 非捕获:
var pattern:RegExp = /(\w+)@(\w+).(?:com|org)/;
var str:String = "bob@example.com";
trace(pattern.exec(str));
  // bob@test.com,bob,example
```

一类特殊的非捕获组是 “向前查找组”，它包括两种类型：“正向前查找组” 和 “负向前查找组”。

使用 (?= 和 ) 定义正向前查找组，它指定组中的子模式位置必须匹配。但是，匹配正向前查找组的字符串部分可能匹配正则表达式中的剩余模式。例如，由于 (?=e) 在下列代码中是正向前查找组，它匹配的字符 e 可以被正则表达式的后续部分匹配，在本例中为捕获组 \w*)：

```
var pattern:RegExp = /sh(?=e)(\w*)/i;
var str:String = "Shelly sells seashells by the seashore";
trace(pattern.exec(str));
// Shelly,elly
```

使用 (?! 和 ) 定义负向前查找组，它指定该组中的子模式位置必须不匹配。例如：

```
var pattern:RegExp = /sh(?!e)(\w*)/i;
var str:String = "She sells seashells by the seashore";
trace(pattern.exec(str));
// shore,ore
```

## 使用命名组

命名组是正则表达式中给定命名标识符的一类组。使用 (?P<name> 和 ) 可定义命名组。例如，下面的正则表达式包含标识符命名为 digits 的命名组：

```
var pattern = /[a-z]+(?P<digits>\d+)[a-z]+/;
```

如果使用 exec() 方法，将添加一个匹配的命名组作为 result 数组属性：

```
var myPattern:RegExp = /([a-z]+)(?P<digits>\d+)[a-z]+/;
var str:String = "a123bcd";
var result:Array = myPattern.exec(str);
trace(result.digits); // 123
```

这里还有一个例子，它使用两个命名组，标识符分别为 name 和 dom：

```
var emailPattern:RegExp =
  /(?P<name>(\w|[_.\-])+)@(?P<dom>((\w|-)+))+\.\w{2,4}+/;
var address:String = "bob@example.com";
var result:Array = emailPattern.exec(address);
trace(result.name); // bob
trace(result.dom); // 示例
```

**注意** 命名组不属于 ECMAScript 语言规范。它们是 ActionScript 3.0 中的新增功能。

## 标志和属性

下表列出了可以为正则表达式设置的五种标志。每种标志都可以作为正则表达式对象属性进行访问。

| 标志 | 属性 | 描述 |
|---|---|---|
| g | global | 匹配多个匹配。 |
| i | ignoreCase | 不区分大小写的匹配。应用于 A-Z 和 a-z 字符，但不能应用于扩展字符，如 É 和 é。 |
| m | multiline | 设置此标志后，$ 和 ^ 可以分别匹配行的开头和结尾。 |
| s | dotall | 设置此标志后，.（点）可以匹配换行符 (\n)。 |
| x | extended | 允许扩展的正则表达式。您可以在正则表达式中键入空格，它将作为模式的一部分被忽略。这可使您更加清晰可读地键入正则表达式代码。 |

请注意这些属性都是只读属性。您在设置正则表达式变量时，可以设置标志（g、i、m、s 和 x），如下所示：

```
var re:RegExp = /abc/gimsx;
```

但是，您无法直接设置命名属性。例如，下列代码将导致错误：

```
var re:RegExp = /abc/;
re.global = true; // 这会产生错误。
```

默认情况下，除非您在正则表达式声明中指定这些标志，否则不会设置，并且相应的属性也会设置为 false。

另外，还有其它两种正则表达式属性：

- lastIndex 属性指定字符串中的索引位置以用于下次调用正则表达式的 exec() 或 test() 方法。
- source 属性指定定义正则表达式的模式部分的字符串。

## g (global) 标志

如果不设置 g (global) 标志，正则表达式匹配将不超过一个。例如，如果正则表达式中不包含 g 标志，String.match() 方法只返回一个匹配的子字符串：

```
var str:String = "she sells seashells by the seashore.";
var pattern:RegExp = /sh\w*/;
trace(str.match(pattern)) // 输出: she
```

如果设置 g 标志，Sting.match() 方法将返回多个匹配，如下所示：

```
var str:String = "she sells seashells by the seashore.";
var pattern:RegExp = /sh\w*/g;
// 模式相同，但是这次设置了 g 标志。
trace(str.match(pattern)); // 输出: she,shells,shore
```

## i (ignoreCase) 标志

默认情况下，正则表达式匹配区分大小写。如果设置 i (ignoreCase) 标志，将忽略区分大小写。例如，正则表达式中的小写字母 s 不会匹配大写字母 S （字符串中的第一个字符）：

```
var str:String = "She sells seashells by the seashore.";
trace(str.search(/sh/)); // 输出: 13 -- 不是第一个字符
```

但是如果设置 i 标志，正则表达式将匹配大写字母 S：

```
var str:String = "She sells seashells by the seashore.";
trace(str.search(/sh/i)); // 输出: 0
```

i 标志仅忽略 A-Z 和 a-z 字符的大小写，而不忽略扩展字符的大小写，如 É 和 é。

## m (multiline) 标志

如果未设置 m (multiline) 标志，^ 将匹配字符串的开头，而 $ 匹配字符串的结尾。如果设置 m 标志，这些字符将分别匹配行的开头和结尾。请考虑使用下列包含换行符的字符串：

```
var str:String = "Test\n";
str += "Multiline";
trace(str.match(/^\w*/g)); // 匹配字符串开头的单词。
```

即使在正则表达式中设置 g (global) 标志，match() 方法也会只匹配一个子字符串，因为对于 ^ （字符串开头）只有一个匹配。输出结果为：

```
Test
```

下面是设置了 m 标志的同一段代码：

```
var str:String = "Test\n";
str += "Multiline";
trace(str.match(/^\w*/gm)); // 匹配每行开头的单词。
```

这次输出结果同时包含两行开头的单词：

```
Test,Multiline
```

请注意，只有 \n 字符表示行的结束。下列字符不表示行的结束：

- 回车 (\r) 字符
- Unicode 行分隔符 (\u2028) 字符
- Unicode 段分隔符 (\u2029) 字符

## s (dotall) 标志

如果未设置 s (dotall 或 “dot all”) 标志，则正则表达式中的点 (.) 将不匹配换行符 (\n)。因此，下面的示例没有匹配：

```
var str:String = "<p>Test\n";
str += "Multiline</p>";
var re:RegExp = /<p>.*?<\/p>/;
trace(str.match(re));
```

但是，如果设置了 s 标志，点就匹配换行符：

```
var str:String = "<p>Test\n";
str += "Multiline</p>";
var re:RegExp = /<p>.*?<\/p>/s;
trace(str.match(re));
```

在本例中，匹配内容是 <p> 标记内的整个子字符串，其中包括换行符：

```
<p>Test
Multiline</p>
```

## x (extended) 标志

正则表达式有时很难阅读，特别是当其包含很多元字符和元序列时。例如：

```
/<p(>|(\s*[^>]*>)).*?<\/p>/gi
```

在正则表达式中使用 x (extended) 标志时，则会忽略在模式中键入的所有空格。例如，下面的正则表达式同前面的示例相同：

```
/     <p    (>   |  (\s*  [^>]*  >))     .*?    <\/p>  /gix
```

如果设置了 x 标志，而且希望匹配空格字符，则应在空格前加上反斜杠。例如，以下两个正则表达式是等效的：

```
/foo bar/
/foo \ bar/x
```

## lastIndex 属性

lastIndex 属性在字符串中指定开始进行下一次搜索的索引位置。对于将 g 标志设置为 true 的正则表达式，此属性会影响对该表达式调用的 exec() 和 test() 方法。例如，请考虑使用以下代码：

```
var pattern:RegExp = /p\w*/gi;
var str:String = "Pedro Piper picked a peck of pickled peppers.";
trace(pattern.lastIndex);
var result:Object = pattern.exec(str);
while (result != null)
{
  trace(pattern.lastIndex);
  result = pattern.exec(str);
}
```

默认情况下，将 lastIndex 属性设置为 0（从字符串的开头开始搜索）。每次匹配完成后，都会设为该匹配后的索引位置。因此，前面代码的输出如下所示：

```
0
5
11
18
25
36
44
```

如果将 global 标志设置为 false，则 exec() 和 test() 方法不会使用或设置 lastIndex 属性。

String 类的 match()、replace() 和 search() 方法都从字符串的开头进行搜索，而不考虑调用该方法时所使用的正则表达式中的 lastIndex 属性设置。（但是，match() 方法不会将 lastIndex 设为 0。）

可以设置 lastIndex 属性来调整正则表达式匹配时字符串中的起始位置。

## source 属性

source 属性指定用于定义正则表达式的模式部分的字符串。例如：

```
var pattern:RegExp = /foo/gi;
trace(pattern.source); // foo
```

# 对字符串使用正则表达式的方法

RegExp 类包含两个方法：exec() 和 test()。

除 RegExp 类的 exec() 和 test() 方法外，String 类还包含以下方法，使您可以在字符串中匹配正则表达式：match()、replace()、search() 和 splice()。

## test() 方法

RegExp 类的 test() 方法只检查提供的字符串是否包含正则表达式的匹配内容，如下例所示：

```
var pattern:RegExp = /Class-\w/;
var str = "Class-A";
trace(pattern.test(str)); // 输出：true
```

## exec() 方法

RegExp 类的 exec() 方法检查提供的字符串是否有正则表达式的匹配，并返回具有如下内容的数组：

- 匹配的子字符串
- 同正则表达式中的任意括号组匹配的子字符串

该数组还包含 index 属性，此属性指明子字符串匹配起始的索引位置。

例如，请考虑使用以下代码：

```
var pattern:RegExp = /\d{3}\-\d{3}-\d{4}/; // 美国电话号码
var str:String = "phone: 415-555-1212";
var result:Array = pattern.exec(str);
trace(result.index, " - ", result);
// 7  -  415-555-1212
```

在正则表达式设置了 g (global) 标志时，多次使用 exec() 方法可以匹配多个子字符串：

```
var pattern:RegExp = /\w*sh\w*/gi;
var str:String = "She sells seashells by the seashore";
var result:Array = pattern.exec(str);

while (result != null)
{
  trace(result.index, "\t", pattern.lastIndex, "\t", result);
  result = pattern.exec(str);
}
// 输出:
// 0  3  She
// 10  19  seashells
// 27  35  seashore
```

## 使用 RegExp 参数的 String 方法

下列 String 类方法将正则表达式作为参数：match()、replace()、search() 和 split()。有关这些方法的详细信息，请参阅第 179 页的 “在字符串中查找模式并替换子字符串”。

# 示例：Wiki 分析器

这个简单的 Wiki 文本转换示例说明了正则表达式的一些用途：

- 将与源 Wiki 模式匹配的文本行转换为相应的 HTML 输出字符串。
- 使用正则表达式将 URL 模式转换为 HTML `<a>` 超链接标记。
- 使用正则表达式将美元符号字符串（如 `"$9.95"`）转换为欧元符号字符串（如 `"8.24 €"`）。

要获取该范例的应用程序文件，请访问 www.adobe.com/go/learn_programmingAS3samples_flash_cn。 WikiEditor 应用程序文件位于文件夹 Samples/WikiEditor 中。该应用程序包含以下文件：

| 文件 | 描述 |
| --- | --- |
| WikiEditor.mxml<br>或<br>WikiEditor.fla | Flash 或 Flex 中的主应用程序文件 （分别为 FLA 和 MXML）。 |
| com/example/programmingas3/<br>regExpExamples/WikiParser.as | 该类包含使用正则表达式将 Wiki 输入文本模式转换为等效 HTML 输出的方法。 |
| com/example/programmingas3/<br>regExpExamples/URLParser.as | 该类包含使用正则表达式将 URL 字符串转换为 HTML `<a>` 超链接标记的方法。 |
| com/example/programmingas3/<br>regExpExamples/CurrencyConverter.as | 该类包含使用正则表达式将美元符号字符串转换为欧元符号字符串的方法。 |

## 定义 WikiParser 类

WikiParser 类包含将 Wiki 输入文本转换为等效 HTML 输出的方法。它虽然不是功能非常强大的 Wiki 转换应用程序，但是说明了正则表达式在模式匹配和字符串转换方面的一些很好的用法。

构造函数和 `setWikiData()` 方法一起，可简单地初始化 Wiki 输入文本范例字符串，如下所示：

```
public function WikiParser()
{
  wikiData = setWikiData();
}
```

当用户单击范例应用程序中的 “Test”按钮时，应用程序将调用 WikiParser 对象的 parseWikiString() 方法。此方法可调用许多其它方法，这些方法再组合输出的 HTML 字符串。

```
public function parseWikiString(wikiString:String):String
{
  var result:String = parseBold(wikiString);
  result = parseItalic(result);
  result = linesToParagraphs(result);
  result = parseBullets(result);
  return result;
}
```

所调用的每个方法 （parseBold()、parseItalic()、linesToParagraphs() 和 parseBullets()）都会使用字符串的 replace() 方法替换由正则表达式定义的匹配模式，以便将 Wiki 输入文本转换为 HTML 格式的文本。

## 转换粗体和斜体模式

parseBold() 方法会查找 Wiki 粗体文本模式 （如 '''foo'''）并将其转换为等效的 HTML 格式 （如 <b>foo</b>），如下所示：

```
private function parseBold(input:String):String
{
  var pattern:RegExp = /'''(.*?)'''/g;
  return input.replace(pattern, "<b>$1</b>");
}
```

请注意，正则表达式的 (.?*) 部分匹配两个定义 ''' 模式之间任意数目的字符 (*)。? 数量表示符使匹配不再是无限制的，因此对于字符串 '''aaa''' bbb '''ccc'''，第一个匹配的字符串是 '''aaa''' 而不是以 ''' 模式开头和结尾的整个字符串。

正则表达式中的括号定义捕获组， replace() 方法通过在替换字符串中使用 $1 代码引用此组。正则表达式中的 g (global) 标志确保 replace() 方法替换字符串中的所有匹配 （不仅仅是第一个）。

parseItalic() 方法的原理与 parseBold() 方法类似，只是前者检查作为斜体文本分隔符的两个省略号 ('') （而不是三个）：

```
private function parseItalic(input:String):String
{
  var pattern:RegExp = /''(.*?)''/g;
  return input.replace(pattern, "<i>$1</i>");
}
```

## 转换项目符号模式

如下例所示，parseBullet() 方法会查找 Wiki 项目符号行模式（如 * foo）并将其转换为等效的 HTML 格式（如 <li>foo</li>）：

```
private function parseBullets(input:String):String
{
  var pattern:RegExp = /^\*(.*)/gm;
  return input.replace(pattern, "<li>$1</li>");
}
```

正则表达式开头的 ^ 符号匹配行的开头。正则表达式中的 m (multiline) 标志使正则表达式用 ^ 符号来匹配行的开头，而不是简单地匹配字符串的开头。

* 模式匹配星号字符（反斜杠用于表示星号本身，而不是 * 数量表示符）。

正则表达式中的括号定义捕获组，replace() 方法通过在替换字符串中使用 $1 代码引用此组。正则表达式中的 g (global) 标志确保 replace() 方法替换字符串中的所有匹配（不仅仅是第一个）。

## 转换段落 Wiki 模式

linesToParagraphs() 方法将每行中输入的 Wiki 字符串转换为 HTML <p> 段落标签。该方法中的这些行从输入的 Wiki 字符串中去除空行：

```
var pattern:RegExp = /^$/gm;
var result:String = input.replace(pattern, "");
```

正则表达式中的 ^ 和 $ 符号分别匹配行的开头和结尾。正则表达式中的 m (multiline) 标志使正则表达式用 ^ 符号来匹配行的开头，而不是简单地匹配字符串的开头。

replace() 方法使用空字符串 ("") 替换所有匹配子字符串（空行）。正则表达式中的 g (global) 标志确保 replace() 方法替换字符串中的所有匹配（不仅仅是第一个）。

# 将 URL 转换为 HTML <a> 标记

当用户单击范例应用程序中的“Test”按钮时，如果用户选中了 urlToATag 复选框，应用程序将调用 URLParser.urlToATag() 静态方法以将 URL 字符串从输入的 Wiki 字符串转换为 HTML <a> 标记。

```
var protocol:String = "((?:http|ftp)://)";
var urlPart:String = "([a-z0-9_-]+\.[a-z0-9_-]+)";
var optionalUrlPart:String = "(\.[a-z0-9_-]*)";
var urlPattern:RegExp = new RegExp(protocol + urlPart + optionalUrlPart,
                        "ig");
var result:String = input.replace(urlPattern,
                "<a href='$1$2$3'><u>$1$2$3</u></a>");
```

RegExp() 构造函数用于将各组成部分组合成正则表达式 (urlPattern)。这些组成部分是定义正则表达式模式部分的各个字符串。

由 protocol 字符串定义的正则表达式模式的第一部分定义了 URL 协议：http:// 或 ftp://。括号定义了由 ? 符号表示的非捕获组。这意味着括号只是用来定义用于 | 逻辑 “或”模式的组，该组不会匹配 replace() 方法的替换字符串中的逆向引用代码 ($1、$2、$3)。

正则表达式的其它组成部分都会使用捕获组 (由模式中的括号表示)，然后用在 replace() 方法的替换字符串中的逆向引用代码 ($1、$2、$3) 中。

由 urlPart 字符串定义的模式部分匹配下列字符中的“至少”一个：a-z、0-9、_ 或 -。+ 数量表示符表明至少一个字符匹配。\. 表明必需的点 (.) 字符。其余部分匹配至少包含以下字符中的一个的字符串：a-z、0-9、_ 或 -。

由 optionalUrlPart 字符串定义的模式部分匹配 “零个或多个”以下字符：点 (.) 字符后跟任意数目的字母数字字符 (包括 _ 和 -)。* 数量表示符表明零个或多个字符匹配。

调用 replace() 方法可应用正则表达式，并且使用逆向引用来组合替换 HTML 字符串。

然后 urlToATag() 方法会调用 emailToATag() 方法，使用类似的技术用 HTML <a> 超链接字符串替换电子邮件模式。在本范例文件中用于匹配 HTTP、FTP 和电子邮件 URL 的正则表达式是相当简单的，只是起到示范作用，还有更复杂的正则表达式更准确地匹配这类 URL。

## 将美元符号字符串转换为欧元符号字符串

当用户单击范例应用程序中的 “Test”按钮时，如果用户选中了 dollarToEuro 复选框，应用程序将调用 CurrencyConverter.usdToEuro() 静态方法以将美元符号字符串 (如 "$9.95") 转换为欧元符号字符串 (如 "8.24 €")，如下所示：

```
var usdPrice:RegExp = /\$([\d,]+.\d+)+/g;
return input.replace(usdPrice, usdStrToEuroStr);
```

第一行定义的简单模式匹配美元符号字符串。请注意，$ 字符前面加反斜杠 (\) 转义字符。

replace() 方法将正则表达式用作模式匹配参数，并调用 usdStrToEuroStr() 函数以确定替换字符串 (欧元值)。

如果将函数名称用作 replace() 方法的第二个参数时，则会将以下内容作为参数传递给被调用的函数：

- 字符串的匹配部分。
- 任何捕获的括号组匹配。按这种方式传递的参数数目因捕获的括号组匹配的数目而异。您可以通过检查函数代码中的 arguments.length - 3 来确定捕获的括号组匹配的数目。
- 字符串中匹配开始的索引位置。
- 完整的字符串。

usdStrToEuroStr() 方法将美元符号字符串模式转换为欧元符号字符串，如下所示：

```
private function usdToEuro(...args):String
{
    var usd:String = args[1];
    usd = usd.replace(",", "");
    var exchangeRate:Number = 0.828017;
    var euro:Number = Number(usd) * exchangeRate;
    trace(usd, Number(usd), euro);
    const euroSymbol:String = String.fromCharCode(8364); // €
    return euro.toFixed(2) + " " + euroSymbol;
}
```

请注意，args[1] 表示由 usdPrice 正则表达式匹配的捕获括号组。这是美元符号字符串的数字部分，也就是没有 $ 符号的美元数目。该方法应用汇率转换并返回生成的字符串（带有尾随符号 €，而不是前导符号 $）。

第 10 章

# 处理事件

10

利用事件处理系统，程序员可以方便地响应用户输入和系统事件。ActionScript 3.0 事件模型不仅方便，而且符合标准，它与 Adobe Flash Player 9 显示列表完美集成在一起。新的事件模型基于文档对象模型 (DOM) 第 3 级事件规范，是业界标准的事件处理体系结构，为 ActionScript 程序员提供了强大而直观的事件处理工具。

本章分为五部分。前两部分提供有关 ActionScript 中的事件处理的背景信息。最后三部分介绍了支持该事件模型的主要概念：事件流、事件对象和事件侦听器。ActionScript 3.0 事件处理系统与显示列表密切交互。本章假定您对显示列表有基本的了解。有关详细信息，请参阅第 319 页的 “显示编程”。

## 目录

# 事件处理基础知识

## 事件处理简介

您可以将事件视为 SWF 文件中发生的程序员感兴趣的任何类型的事件。例如，大多数 SWF 文件都支持某些类型的用户交互，无论是像响应鼠标单击这样简单的用户交互，还是像接受和处理表单中输入的数据这样复杂的用户交互。与 SWF 文件进行的任何此类用户交互都可以视为事件。也可能会在没有任何直接用户交互的情况下发生事件，例如，从服务器加载完数据或者连接的摄像头变为活动状态时。

在 ActionScript 3.0 中，每个事件都由一个事件对象表示。事件对象是 Event 类或其某个子类的实例。事件对象不但存储有关特定事件的信息，还包含便于操作事件对象的方法。例如，当 Flash Player 检测到鼠标单击时，它会创建一个事件对象（MouseEvent 类的实例）以表示该特定鼠标单击事件。

创建事件对象之后， Flash Player 即 "调度" 该事件对象，这意味着将该事件对象传递给作为事件目标的对象。作为所调度事件对象的目标的对象称为 "事件目标"。例如，当连接的摄像头变为活动状态时， Flash Player 会向事件目标直接调度一个事件对象，此时，该事件对象就是代表摄像头的对象。但是，如果事件目标在显示列表中，则在显示列表层次结构中将事件对象向下传递，直到到达事件目标为止。在某些情况下，该事件对象随后会沿着相同路线在显示列表层次结构中向上 "冒泡" 回去。显示列表层次结构中的这种遍历行为称为 "事件流"。

您可以使用事件侦听器 "侦听" 代码中的事件对象。"事件侦听器" 是您编写的用于响应特定事件的函数或方法。要确保您的程序响应事件，必须将事件侦听器添加到事件目标，或添加到作为事件对象事件流的一部分的任何显示列表对象。

无论何时编写事件侦听器代码，该代码都会采用以下基本结构 (以粗体显示的元素是占位符，您将针对具体情况对其进行填写)：

```
function eventResponse(eventObject:EventType):void
{
    // 此处是为响应事件而执行的动作。
}

eventTarget.addEventListener(EventType.EVENT_NAME, eventResponse);
```

此代码执行两个操作。首先，它定义一个函数，这是指定为响应事件而执行的动作的方法。接下来，调用源对象的 `addEventListener()` 方法，实际上就是为指定事件 "订阅" 该函数，以便当该事件发生时，执行该函数的动作。当事件实际发生时，事件目标将检查其注册为事件侦听器的所有函数和方法的列表。然后，它依次调用每个对象，以将事件对象作为参数进行传递。

您需要在此代码中更改四项内容以创建自己的事件侦听器。第一，必须将函数名称更改为要使用的名称（必须在两个位置更改此内容，代码将在此处显示 `eventResponse`）。第二，必须为要侦听的事件（代码中的 `EventType`）所调度的事件对象指定相应的类名称，并且必须为特定事件（列表中的 `EVENT_NAME`）指定相应的常量。第三，必须针对调度事件（此代码中的 `eventTarget`）的对象调用 `addEventListener()` 方法。您可以选择更改用作函数参数（此代码中的 `eventObject`）的变量的名称。

## 常见事件处理任务

下面是常见的事件处理任务，本章将介绍其中的每项任务：

- 编写代码以响应事件
- 阻止代码响应事件
- 处理事件对象
- 处理事件流：
  - 识别事件流信息
  - 停止事件流
  - 禁止默认行为
- 从类中调度事件
- 创建自定义事件类型

## 重要概念和术语

以下参考列表包含将会在本章中遇到的重要术语：

- 默认行为 (Default behavior)：某些事件包含通常与事件一起发生的行为（称为默认行为）。例如，当用户在文本字段中键入文本时，将引发文本输入事件。该事件的默认行为是实际显示在文本字段中键入的字符，但您可以覆盖该默认行为（如果由于某种原因，您不希望显示键入的字符）。
- 调度 (Dispatch)：通知事件侦听器发生了事件。
- 事件 (Event)：对象可以通知其它对象它所发生的情况。
- 事件流 (Event flow)：如果显示列表中的对象（屏幕上显示的对象）发生事件，则会向包含该对象的所有对象通知此事件，并依次通知其事件侦听器。此过程从舞台开始，并在显示列表中一直进行到发生事件的实际对象，然后再返回到舞台。此过程称为事件流。
- 事件对象 (Event object)：此对象包含发生的特定事件的相关信息，当调度事件时，此信息将被发送到所有侦听器。

- 事件目标 (Event target)：实际调度事件的对象。例如，如果用户单击位于 Sprite （位于舞台内）内的按钮，所有这些对象将调度事件，但事件目标是指实际发生事件的对象，此处指单击的按钮。
- 侦听器 (Listener)：对象或在对象中注册其自身的函数，用于指示发生特定事件时应通知它。

## 完成本章中的示例

学习本章的过程中，您可能想要自己动手测试一些示例代码清单。本章中的几乎所有代码清单都包括一个 `trace()` 函数调用，用于测试代码的结果。要测试本章中的代码清单，请执行以下操作：

1. 创建一个空的 Flash 文档。
2. 在时间轴上选择一个关键帧。
3. 打开 "动作" 面板，将代码清单复制到 "脚本" 窗格中。
4. 使用 "控制" > "测试影片" 运行程序。

   您将在 "输出" 面板中看到该代码清单的 `trace` 函数的结果。

某些代码清单更为复杂一些，并且编写为类的形式。要测试这些示例，请执行以下操作：

1. 创建一个空的 Flash 文档并将它保存到您的计算机上。
2. 创建一个新的 ActionScript 文件，并将它保存到 Flash 文档所在的目录中。文件名应与代码清单中的类的名称一致。例如，如果代码清单定义一个名为 EventTest 的类，则使用名称 EventTest.as 来保存 ActionScript 文件。
3. 将代码清单复制到 ActionScript 文件中并保存该文件。
4. 在 Flash 文档中，单击舞台或工作区的空白部分，以激活文档的 "属性" 检查器。
5. 在 "属性" 检查器的 "文档类" 字段中，输入您从文本中复制的 ActionScript 类的名称。
6. 使用 "控制" > "测试影片" 运行程序。

   您将在 "输出" 面板中看到该示例的结果。

测试示例代码清单的这些技术在第 53 页的 "测试本章内的示例代码清单"中有更详细的介绍。

# ActionScript 3.0 事件处理与早期版本事件处理的不同之处

ActionScript 3.0 中的事件处理与早期 ActionScript 版本中的事件处理之间的一个最显著的区别是：在 ActionScript 3.0 中，只有一个事件处理系统，而在早期的 ActionScript 版本中，则有几个不同的事件处理系统。本部分先概述早期 ActionScript 版本中的事件处理的工作原理，然后讨论 ActionScript 3.0 中的事件处理的变化情况。

## 早期 ActionScript 版本中的事件处理

ActionScript 3.0 之前的 ActionScript 版本中提供许多不同的方法来处理事件：

- `on()` 事件处理函数，可以直接放在 Button 和 MovieClip 实例上
- `onClipEvent()` 处理函数，可以直接放在 MovieClip 实例上
- 回调函数属性，例如 `XML.onload` 和 `Camera.onActivity`
- 使用 `addListener()` 方法注册的事件侦听器
- 部分实现了 DOM 事件模型的 UIEventDispatcher 类

其中的每一种机制都有其自己的若干优点和局限性。`on()` 和 `onClipEvent()` 处理函数易于使用，但使随后对项目的维护变得较为困难，因为很难查找直接放在按钮和影片剪辑上的代码。回调函数也很容易实现，但对于任何指定事件，仅限于使用一个回调函数。事件侦听器较难实现：它们不但要求创建侦听器对象和函数，而且要求向生成事件的对象注册侦听器。这虽然增加了开销，但您可以创建若干侦听器对象，并针对同一个事件注册这些对象。

对 ActionScript 2.0 组件的开发形成了另一个事件模型。该新模型包含在 UIEventDispatcher 类中，并且基于 DOM 事件规范的子集。熟悉组件事件处理的开发人员将会发现过渡到新的 ActionScript 3.0 事件模型相对来说较为容易。

遗憾的是，各个事件模型使用的语法以不同的方式相互重叠，并且在其它方面各自不同。例如，在 ActionScript 2.0 中，某些属性（例如 `TextField.onChanged`）可用作回调函数或事件侦听器。但是，根据您是否在使用支持侦听器或六个类之一的 UIEventDispatcher 类，用于注册侦听器对象的语法有所不同。对于 Key、Mouse、MovieClipLoader、Selection、Stage 和 TextField 类；请使用 `addListener()` 方法，但对于组件事件处理，请使用名为 `addEventListener()` 的方法。

不同事件处理模型所导致的另一个复杂性是：根据所使用的机制的不同，事件处理函数的范围大不相同。也就是说，关键字 `this` 的含义在各个事件处理系统中并不一致。

## ActionScript 3.0 中的事件处理

ActionScript 3.0 引入了单一事件处理模型，以替代以前各语言版本中存在的众多不同的事件处理机制。该新事件模型基于文档对象模型 (DOM) 第 3 级事件规范。虽然 SWF 文件格式并不专门遵循文档对象模型标准，但显示列表和 DOM 结构之间存在的相似性足以使 DOM 事件模型的实现成为可能。显示列表中的对象类似于 DOM 层次结构中的节点，在本讨论中，术语 “显示列表对象”和 “节点”可互换使用。

Flash Player 实现的 DOM 事件模型包括一个名为 “默认行为”的概念。“默认行为”是 Flash Player 作为特定事件的正常后果而执行的操作。

### 默认行为

开发人员通常负责编写响应事件的代码。但在某些情况下，行为通常与某一事件关联，使得 Flash Player 会自动执行该行为，除非开发人员添加了取消该行为的代码。由于 Flash Player 会自动表现该行为，因此这类行为称为默认行为。

例如，当用户在 TextField 对象中输入文本时，普遍期待文本将在该 TextField 对象中显示，因此该行为被内置到 Flash Player 中。如果您不希望该默认行为发生，可以使用新的事件处理系统来取消它。当用户在 TextField 对象中输入文本时，Flash Player 创建 TextEvent 类的实例以表示该用户输入。为阻止 Flash Player 显示 TextField 对象中的文本，必须访问该特定 TextEvent 实例并调用该实例的 `preventDefault()` 方法。

并非所有默认行为都可以阻止。例如，当用户双击 TextField 对象中的单词时，Flash Player 会生成一个 MouseEvent 对象。无法阻止的默认行为是：鼠标点击的单词突出显示。

许多类型的事件对象没有关联的默认行为。例如，当建立网络连接时， Flash Player 调度一个连接事件对象，但没有与该对象关联的默认行为。Event 类及其子类的 API 文档列出了每一类型的事件，并说明所有关联的默认行为，以及是否可以阻止该行为。

默认行为仅与由 Flash Player 调度的事件对象关联，但通过 ActionScript 以编程方式调度的事件对象则不存在默认行为。了解这一点很重要。例如，可以使用 EventDispatcher 类的方法来调度类型为 `textInput` 的事件对象，但该事件对象没有关联的默认行为。也就是说， Flash Player 不会因为您以编程方式调度了 `textInput` 事件而在 TextField 对象中显示字符。

## ActionScript 3.0 中的事件侦听器的新增功能

对于使用过 ActionScript 2.0 `addListener()` 方法的开发人员来说，指出 ActionScript 2.0 事件侦听器模型和 ActionScript 3.0 事件模型之间的差别可能会有所帮助。下表说明两个事件模型之间的几个主要差别：

- 要在 ActionScript 2.0 中添加事件侦听器，请在某些情况下使用 `addListener()`，在其它情况下使用 `addEventListener()`；而在 ActionScript 3.0 中，则始终使用 `addEventListener()`。
- ActionScript 2.0 中没有事件流，这意味着，只能对广播事件的对象调用 `addListener()` 方法；而在 ActionScript 3.0 中，可以对属于事件流一部分的任何对象调用 `addEventListener()` 方法。
- 在 ActionScript 2.0 中，事件侦听器可以是函数、方法或对象，而在 ActionScript 3.0 中，只有函数或方法可以是事件侦听器。

# 事件流

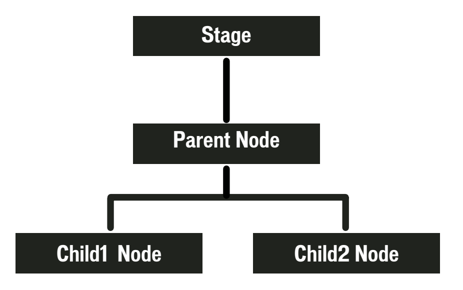
只要发生事件， Flash Player 就会调度事件对象。如果事件目标不在显示列表中，则 Flash Player 将事件对象直接调度到事件目标。例如，Flash Player 将 progress 事件对象直接调度到 URLStream 对象。但是，如果事件目标在显示列表中，则 Flash Player 将事件对象调度到显示列表，事件对象将在显示列表中穿行，直到到达事件目标。

“事件流”说明事件对象如何在显示列表中穿行。显示列表以一种可以描述为树的层次结构形式进行组织。位于显示列表层次结构顶部的是舞台，它是一种特殊的显示对象容器，用作显示列表的根。舞台由 flash.display.Stage 类表示，且只能通过显示对象访问。每个显示对象都有一个名为 `stage` 的属性，该属性表示应用程序的舞台。

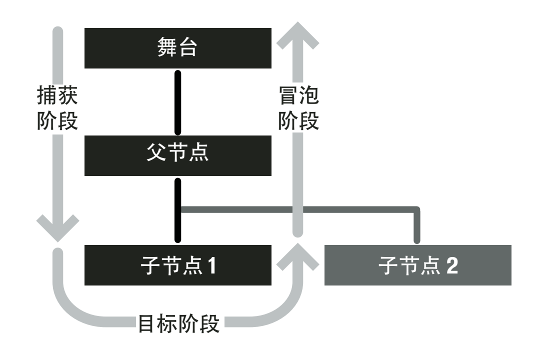
当 Flash Player 调度事件对象时，该事件对象进行一次从舞台到 “目标节点”的往返行程。DOM 事件规范将目标节点定义为代表事件目标的节点。也就是说，目标节点是发生了事件的显示列表对象。例如，如果用户单击名为 `child1` 的显示列表对象， Flash Player 将使用 `child1` 作为目标节点来调度事件对象。

从概念上来说，事件流分为三部分。第一部分称为捕获阶段，该阶段包括从舞台到目标节点的父节点范围内的所有节点。第二部分称为目标阶段，该阶段仅包括目标节点。第三部分称为冒泡阶段。冒泡阶段包括从目标节点的父节点返回到舞台的行程中遇到的节点。

如果您将显示列表想像为一个垂直的层次结构，其中舞台位于顶层（如下图显示），那么这些阶段的名称就更容易理解了：




如果用户单击 `Child1`，Flash Player 将向事件流调度一个事件对象。如下面的图像所示，对象的行程从 `Stage` 开始，向下移动到 `Parent`，然后移动到 `Child1`，再“冒泡”返回到 `Stage`：在行程中重新经过 `Parent`，再返回到 `Stage`。




在该示例中，捕获阶段在首次向下行程中包括 `Stage` 和 `Parent`。目标阶段包括在 `Child1` 花费的时间。冒泡阶段包括在向上返回到根节点的行程中遇到的 `Parent` 和 `Stage`。

事件流使现在的事件处理系统比 ActionScript 程序员以前使用的事件处理系统功能更为强大。早期版本的 ActionScript 中没有事件流，这意味着事件侦听器只能添加到生成事件的对象。在 ActionScript 3.0 中，您不但可以将事件侦听器添加到目标节点，还可以将它们添加到事件流中的任何节点。

当用户界面组件包含多个对象时，沿事件流添加事件侦听器的功能十分有用。例如，按钮对象通常包含一个用作按钮标签的文本对象。如果无法将侦听器添加到事件流，您将必须将侦听器添加到按钮对象和文本对象，以确保您收到有关在按钮上任何位置发生的单击事件的通知。而事件流的存在则使您可以将一个事件侦听器放在按钮对象上，以处理文本对象上发生的单击事件或按钮对象上未被文本对象遮住的区域上发生的单击事件。

不过，并非每个事件对象都参与事件流的所有三个阶段。某些类型的事件（例如 `enterFrame` 和 `init` 类型的事件）会直接调度到目标节点，并不参与捕获阶段和冒泡阶段。其它事件可能以不在显示列表中的对象为目标，例如调度到 Socket 类的实例的事件。这些事件对象也将直接流至目标对象，而不参与捕获和冒泡阶段。

要查明特定事件类型的行为，可以查看 API 文档或检查事件对象的属性。下面的部分介绍了如何检查事件对象的属性。

# 事件对象

在新的事件处理系统中，事件对象有两个主要用途。首先，事件对象通过将特定事件的信息存储在一组属性中，来代表实际事件。第二，事件对象包含一组方法，可用于操作事件对象和影响事件处理系统的行为。

为方便对这些属性和方法的访问，Flash Player API 定义了一个 Event 类，作为所有事件对象的基类。 Event 类定义一组基本的适用于所有事件对象的属性和方法。

本部分首先讨论 Event 类属性，然后介绍 Event 类方法，最后说明 Event 类的子类存在的意义。

## 了解 Event 类属性

Event 类定义许多只读属性和常数，以提供有关事件对象的重要信息。以下内容尤其重要：

- 事件对象类型由常数表示，并存储在 `Event.type` 属性中。
- 事件的默认行为是否可以被阻止由布尔值表示，并存储在 `Event.cancelable` 属性中。
- 事件流信息包含在其余属性中。

### 事件对象类型

每个事件对象都有关联的事件类型。数据类型以字符串值的形式存储在 `Event.type` 属性中。知道事件对象的类型是非常有用的，这样您的代码就可以区分不同类型的事件。例如，下面的代码指定 `clickHandler()` 侦听器函数应响应传递给 `myDisplayObject` 的任何鼠标单击事件对象。

```
myDisplayObject.addEventListener(MouseEvent.CLICK, clickHandler);
```

大约有 20 多种事件类型与 Event 类自身关联并由 Event 类常数表示，其中某些数据类型显示在摘自 Event 类定义的以下代码中：

```
package flash.events
{
  public class Event
  {
    // 类常数
    public static const ACTIVATE:String = "activate";
    public static const ADDED:String    = "added";
    // 为简便起见，省略了其余常数
  }
}
```

这些常数提供了引用特定事件类型的简便方法。您应使用这些常数而不是它们所代表的字符串。如果您的代码中拼错了某个常数名称，编译器将捕获到该错误，但如果您改为使用字符串，则编译时可能不会出现拼写错误，这可能导致难以调试的意外行为。例如，添加事件侦听器时，使用以下代码：

```
myDisplayObject.addEventListener(MouseEvent.CLICK, clickHandler);
```

而不是使用：

```
myDisplayObject.addEventListener("click", clickHandler);
```

## 默认行为信息

代码可通过访问 `cancelable` 属性来检查是否可以阻止任何指定事件对象的默认行为。`cancelable` 属性包含一个布尔值，用于指示是否可以阻止默认行为。您可以使用 `preventDefault()` 方法阻止或取消与少量事件关联的默认行为。有关详细信息，请参阅第 279 页的 “取消默认事件行为”。

## 事件流信息

其余 Event 类属性包含有关事件对象及其与事件流的关系的重要信息，如以下列表所述：

- `bubbles` 属性包含有关事件流中事件对象参与的部分的信息。
- `eventPhase` 属性指示事件流中的当前阶段。
- `target` 属性存储对事件目标的引用。
- `currentTarget` 属性存储对当前正在处理事件对象的显示列表对象的引用。

## bubbles 属性

如果事件对象参与事件流的冒泡阶段，则将该事件称为“冒泡”，这指的是从目标节点将事件对象往回传递，经过目标节点的父节点，直到到达舞台。Event.bubbles 属性存储一个布尔值，用于指示事件对象是否参与冒泡阶段。由于冒泡的所有事件还参与捕获和目标阶段，因此这些事件参与事件流的所有三个阶段。如果值为 true，则事件对象参与所有三个阶段。如果值为 false，则事件对象不参与冒泡阶段。

## eventPhase 属性

您可以通过调查任何事件对象的 eventPhase 属性来确定事件阶段。eventPhase 属性包含一个无符号整数值，该值代表三个事件流阶段中的一个阶段。Flash Player API 定义了单独的 EventPhase 类，该类包含三个对应于三个无符号整数值的常量，如以下摘录代码中所示：

```
package flash.events
{
  public final class EventPhase
  {
    public static const CAPTURING_PHASE:uint = 1;
    public static const AT_TARGET:uint       = 2;
    public static const BUBBLING_PHASE:uint  = 3;
  }
}
```

这些常数对应于 eventPhase 属性的三个有效值。使用这些常数可以使您的代码可读性更好。例如，如果要确保仅当事件目标在目标阶段中时才调用名为 myFunc() 的函数，您可以使用以下代码来测试此条件：

```
if (event.eventPhase == EventPhase.AT_TARGET)
{
  myFunc();
}
```

## target 属性

target 属性包含对作为事件目标的对象的引用。在某些情况下，这很简单，例如当麦克风变为活动状态时，事件对象的目标是 Microphone 对象。但是，如果目标在显示列表中，则必须考虑显示列表层次结构。例如，如果用户在包括重叠的显示列表对象的某一点输入一个鼠标单击，则 Flash Player 始终会选择距离舞台层次最深的对象作为事件目标。

对于复杂的 SWF 文件，特别是那些通常使用更小的子对象来修饰按钮的 SWF 文件，target 属性可能并不常用，因为它通常指向按钮的子对象，而不是按钮。在这些情况下，常见的做法是将事件侦听器添加到按钮并使用 currentTarget 属性，因为该属性指向按钮，而 target 属性可能指向按钮的子对象。

### currentTarget 属性

`currentTarget` 属性包含对当前正在处理事件对象的对象的引用。您并不知道哪个节点当前正在处理您要检查的事件对象，虽然这似乎很奇怪，但请记住，您可以向该事件对象的事件流中的任何显示对象添加侦听器函数，并且可以将侦听器函数放在任何位置。而且，可以将相同的侦听器函数添加到不同的显示对象。随着项目大小和复杂性的增加，`currentTarget` 属性会变得越来越有用。

# 了解 Event 类方法

共有三种类别的 Event 类方法：

- 实用程序方法：可以创建事件对象的副本或将其转换为字符串
- 事件流方法：用于从事件流中删除事件对象
- 默认行为方法：可阻止默认行为或检查是否已阻止默认行为

## Event 流实用程序方法

Event 类中有两个实用程序方法。`clone()` 方法用于创建事件对象的副本。`toString()` 方法用于生成事件对象属性的字符串表示形式以及它们的值。这两个方法都由事件模型系统在内部使用，但对开发人员公开以用于一般用途。

对于创建 Event 类的子类的高级开发人员来说，必须覆盖和实现两个实用程序方法的版本，以确保事件子类正常使用。

## 停止事件流

可以调用 `Event.stopPropogation()` 方法或 `Event.stopImmediatePropogation()` 方法来阻止在事件流中继续执行事件对象。这两种方法几乎相同，只有在是否允许执行当前节点的其它事件侦听器方面不同：

- `Event.stopPropogation()` 方法可阻止事件对象移动到下一个节点，但只有在允许执行当前节点上的任何其它事件侦听器之后才起作用。
- `Event.stopImmediatePropogation()` 方法也阻止事件对象移动到下一个节点，但不允许执行当前节点上的任何其它事件侦听器。

调用其中任何一个方法对是否发生与事件关联的默认行为没有影响。使用 Event 类的默认行为方法可以阻止默认行为。

## 取消默认事件行为

与取消默认行为有关的两个方法是 `preventDefault()` 方法和 `isDefaultPrevented()` 方法。调用 `preventDefault()` 方法可取消与事件关联的默认行为。要查看是否已针对事件对象调用了 `preventDefault()`，请调用 `isDefaultPrevented()` 方法，如果已经调用，该方法将返回值 `true`，否则返回值 `false`。

仅当可以取消事件的默认行为时，`preventDefault()` 方法才起作用。可通过参考该事件类型的 API 文档或使用 ActionScript 检查事件对象的 `cancelable` 属性来确定是否属于这种情况。

取消默认行为对事件对象通过事件流的进度没有影响。使用 Event 类的事件流方法可以从事件流中删除事件对象。

## Event 类的子类

对于很多事件，Event 类中定义的一组公共属性已经足够了。但是，Event 类中可用的属性无法捕获其它事件具有的独特的特性。 Flash Player API 为这些事件定义了 Event 类的几个子类。

每个子类提供了对于该事件类别来说唯一的附加属性和事件类型。例如，与鼠标输入相关的事件具有若干独特的特性，无法被 Event 类中定义的属性捕获。 MouseEvent 类添加了 10 个属性，扩展了 Event 类。这 10 个属性包含诸如鼠标事件的位置和在鼠标事件过程中是否按下了特定键等信息。

Event 子类还包含代表与子类关联的事件类型的常量。例如， MouseEvent 类定义几种鼠标事件类型的内容，包括 `click`、 `doubleClick`、 `mouseDown` 和 `mouseUp` 事件类型。

正如第 278 页的 “Event 流实用程序方法”一节中所述，创建 Event 子类时，您必须覆盖 `clone()` 和 `toString()` 方法才能提供子类所特有的功能。

# 事件侦听器

事件侦听器也称为事件处理函数，是 Flash Player 为响应特定事件而执行的函数。添加事件侦听器的过程分为两步。首先，为 Flash Player 创建一个为响应事件而执行的函数或类方法。这有时称为侦听器函数或事件处理函数。然后，使用 `addEventListener()` 方法，在事件的目标或位于适当事件流上的任何显示列表对象中注册侦听器函数。

## 创建侦听器函数

创建侦听器函数是 ActionScript 3.0 事件模型与 DOM 事件模型不同的一个方面。在 DOM 事件模型中，事件侦听器和侦听器函数之间有一个明显的不同：即事件侦听器是实现 EventListener 接口的类的实例，而侦听器是该类的名为 `handleEvent()` 的方法。在 DOM 事件模型中，您注册的是包含侦听器函数的类实例，而不是实际的侦听器函数。

在 ActionScript 3.0 事件模型中，事件侦听器和侦听器函数之间没有区别。 ActionScript 3.0 没有 EventListener 接口，侦听器函数可以在类外部定义，也可以定义为类的一部分。此外，无需将侦听器函数命名为 `handleEvent()` — 可以将它们命名为任何有效的标识符。在 ActionScript 3.0 中，您注册的是实际侦听器函数的名称。

## 在类外部定义的侦听器函数

以下代码创建一个显示红色正方形的简单 SWF 文件。名为 `clickHandler()` 的侦听器函数（不是类的一部分）侦听红色正方形上的鼠标单击事件。

```
package
{
  import flash.display.Sprite;

  public class ClickExample extends Sprite
  {
    public function ClickExample()
    {
      var child:ChildSprite = new ChildSprite();
      addChild(child);
    }
  }
}

import flash.display.Sprite;
import flash.events.MouseEvent;

class ChildSprite extends Sprite
{
  public function ChildSprite()
  {
    graphics.beginFill(0xFF0000);
    graphics.drawRect(0,0,100,100);
    graphics.endFill();
    addEventListener(MouseEvent.CLICK, clickHandler);
  }
}

function clickHandler(event:MouseEvent):void
{
  trace("clickHandler detected an event of type: " + event.type);
  trace("the this keyword refers to: " + this);
}
```

当用户通过单击正方形与生成的 SWF 文件交互时， Flash Player 生成以下跟踪输出：

```
clickHandler detected an event of type: click
the this keyword refers to: [object global]
```

请注意，事件对象作为参数传递到 clickHandler()。这就允许您的侦听器函数检查事件对象。在该示例中，使用事件对象的 type 属性来确定该事件是否为单击事件。

该示例还检查 this 关键字的值。在本例中，this 表示全局对象，这是因为函数是在任何自定义类或对象外部定义的。

## 定义为类方法的侦听器函数

下面的示例与前面定义 ClickExample 类的示例相同，只是将 clickHandler() 函数定义为 ChildSprite 类的方法：

```
package
{
  import flash.display.Sprite;

  public class ClickExample extends Sprite
  {
    public function ClickExample()
    {
      var child:ChildSprite = new ChildSprite();
      addChild(child);
    }
  }
}

import flash.display.Sprite;
import flash.events.MouseEvent;

class ChildSprite extends Sprite
{
  public function ChildSprite()
  {
    graphics.beginFill(0xFF0000);
    graphics.drawRect(0,0,100,100);
    graphics.endFill();
    addEventListener(MouseEvent.CLICK, clickHandler);
  }
  private function clickHandler(event:MouseEvent):void
  {
    trace("clickHandler detected an event of type: " + event.type);
    trace("the this keyword refers to: " + this);
  }
}
```

当用户通过单击红色正方形与生成的 SWF 文件交互时，Flash Player 生成以下跟踪输出：

```
clickHandler detected an event of type: click
the this keyword refers to: [object ChildSprite]
```

请注意，this 关键字引用名为 child 的 ChildSprite 实例。这是与 ActionScript 2.0 相比行为上的一处变化。如果您使用过 ActionScript 2.0 中的组件，您可能会记得，将类方法传递给 UIEventDispatcher.addEventListener() 时，方法的作用域被绑定到广播事件的组件，而不是在其中定义侦听器方法的类。也就是说，如果在 ActionScript 2.0 中使用该技术，this 关键字将引用广播事件的组件，而不是 ChildSprite 实例。

这对于某些程序员来说是一个重要问题，因为这意味着他们无法访问包含侦听器方法的类的其它方法和属性。过去，ActionScript 2.0 程序员可以使用 mx.util.Delegate 类更改侦听器方法的作用域以解决该问题。不过，现在已不再需要这样做了，因为 ActionScript 3.0 在调用 addEventListener() 时会创建一个绑定方法。这样，this 关键字引用名为 child 的 ChildSprite 实例，且程序员可以访问 ChildSprite 类的其它方法和属性。

## 不应使用的事件侦听器

还有第三种技术可用于创建一个通用对象，该对象具有指向动态分配的侦听器函数的属性，但不推荐使用该技术。在此讨论这种技术是因为它在 ActionScript 2.0 中经常使用，但在 ActionScript 3.0 中不应使用。不推荐使用该技术的原因是，this 关键字将引用全局对象，而不是您的侦听器对象。

下面的示例与前面的 ClickExample 类示例相同，只是将侦听器函数定义为名为 myListenerObj 的通用对象的一部分：

```
package
{
  import flash.display.Sprite;

  public class ClickExample extends Sprite
  {
    public function ClickExample()
    {
      var child:ChildSprite = new ChildSprite();
      addChild(child);
    }
  }
}

import flash.display.Sprite;
import flash.events.MouseEvent;

class ChildSprite extends Sprite
{
  public function ChildSprite()
  {
    graphics.beginFill(0xFF0000);
    graphics.drawRect(0,0,100,100);
    graphics.endFill();
    addEventListener(MouseEvent.CLICK, myListenerObj.clickHandler);
```

```
  }
}

var myListenerObj:Object = new Object();
myListenerObj.clickHandler = function (event:MouseEvent):void
{
    trace("clickHandler detected an event of type: " + event.type);
    trace("the this keyword refers to: " + this);
}
```

跟踪的结果将类似如下：

```
clickHandler detected an event of type: click
the this keyword refers to: [object global]
```

您可能以为 this 引用 myListenerObj，且跟踪输出应为 [object Object]，但实际上它引用的是全局对象。将动态属性名称作为参数传递到 addEventListener() 时， Flash Player 无法创建绑定方法。这是因为作为 listener 参数传递的只不过是侦听器函数的内存地址，Flash Player 无法将该内存地址与 myListenerObj 实例关联起来。

## 管理事件侦听器

使用 IEventDispatcher 接口的方法来管理侦听器函数。 IEventDispatcher 接口是 ActionScript 3.0 版本的 DOM 事件模型的 EventTarget 接口。虽然名称 IEventDispatcher 似乎暗示着其主要用途是发送（调度）事件对象，但该类的方法实际上更多用于注册、检查和删除事件侦听器。 IEventDispatcher 接口定义五个方法，如以下代码中所示：

```
package flash.events
{
  public interface IEventDispatcher
  {
    function addEventListener(eventName:String,
             listener:Object,
             useCapture:Boolean=false,
             priority:Integer=0,
             useWeakReference:Boolean=false):Boolean;

    function removeEventListener(eventName:String,
           listener:Object,
           useCapture:Boolean=false):Boolean;

    function dispatchEvent(eventObject:Event):Boolean;

    function hasEventListener(eventName:String):Boolean;
    function willTrigger(eventName:String):Boolean;
  }
}
```

Flash Player API 使用 EventDispatcher 类来实现 IEventDispatcher 接口，该类用作可以是事件目标或事件流一部分的所有类的基类。例如， DisplayObject 类继承自 EventDispatcher 类。这意味着，显示列表中的所有对象都可以访问 IEventDispatcher 接口的方法。

## 添加事件侦听器

`addEventListener()` 方法是 IEventDispatcher 接口的主要函数。使用它来注册侦听器函数。两个必需的参数是 `type` 和 `listener`。`type` 参数用于指定事件的类型。`listener` 参数用于指定发生事件时将执行的侦听器函数。`listener` 参数可以是对函数或类方法的引用。




指定 `listener` 参数时，不要使用括号。例如，在下面的 `addEventListener()` 方法调用中，指定 `clickHandler()` 函数时没有使用括号：
`addEventListener(MouseEvent.CLICK, clickHandler)`。

通过使用 `addEventListener()` 方法的 `useCapture` 参数，可以控制侦听器将处于活动状态的事件流阶段。如果 `useCapture` 设置为 `true`，侦听器将在事件流的捕获阶段成为活动状态。如果 `useCapture` 设置为 `false`，侦听器将在事件流的目标阶段和冒泡阶段处于活动状态。要在事件流的所有阶段侦听某一事件，您必须调用 `addEventListener()` 两次，第一次调用时将 `useCapture` 设置为 `true`，第二次调用时将 `useCapture` 设置为 `false`。

`addEventListener()` 方法的 `priority` 参数并不是 DOM Level 3 事件模型的正式部分。ActionScript 3.0 中包括它是为了在组织事件侦听器时提供更大的灵活性。调用 `addEventListener()` 时，可以将一个整数值作为 `priority` 参数传递，以设置该事件侦听器的优先级。默认值为 0，但您可以将它设置为负整数值或正整数值。将优先执行此数字较大的事件侦听器。对于具有相同优先级的事件侦听器，则按它们的添加顺序执行，因此将优先执行较早添加的侦听器。

可以使用 `useWeakReference` 参数来指定对侦听器函数的引用是弱引用还是正常引用。通过将此参数设置为 `true`，可避免侦听器函数在不再需要时仍然存在于内存中的情况。 Flash Player 使用一项称为“垃圾回收”的技术从内存中清除不再使用的对象。如果不存在对某个对象的引用，则该对象被视为不再使用。垃圾回收器不考虑弱引用，这意味着如果侦听器函数仅具有指向它的弱引用，则符合垃圾回收条件。

该参数的一个重要后果与显示对象事件的处理有关。通常，您可能希望从显示列表中删除显示对象时，也将其从内存中删除。但是，如果其它对象已在 `useWeakReference` 参数设置为 `false` （默认值）时作为侦听器订阅该显示对象，该显示对象将继续存在于 Flash Player 的内存中，即使它已不再显示在屏幕中。要解决该问题，可以使所有侦听器在 `useWeakReference` 参数设置为 `true` 时订阅该显示对象，或者使用 `removeEventListener()` 方法从该显示对象中删除所有事件侦听器。

## 删除事件侦听器

可以使用 `removeEventListener()` 方法删除不再需要的事件侦听器。删除将不再使用的所有侦听器是个好办法。必需的参数包括 `eventName` 和 `listener` 参数，这与 `addEventListener()` 方法所需的参数相同。回想一下，您可以通过调用 `addEventListener()` 两次（第一次调用时将 `useCapture` 设置为 `true`，第二次调用时将其设置为 `false`），在所有事件阶段侦听事件。要删除这两个事件侦听器，您需要调用 `removeEventListener()` 两次，第一次调用时将 `useCapture` 设置为 `true`，第二次调用时将其设置为 `false`。

## 调度事件

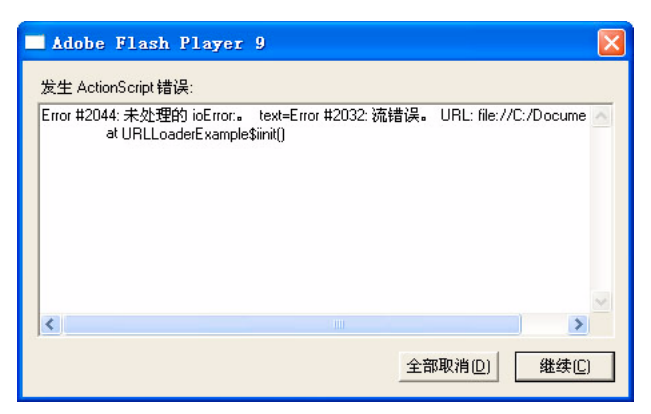
高级程序员可以使用 `dispatchEvent()` 方法将自定义事件对象调度到事件流。该方法唯一接受的参数是对事件对象的引用，此事件对象必须是 Event 类的实例或子类。调度后，事件对象的 `target` 属性将设置为对其调用了 `dispatchEvent()` 的对象。

## 检查现有的事件侦听器

IEventDispatcher 接口的最后两个方法提供有关是否存在事件侦听器的有用信息。如果在特定显示列表对象上发现特定事件类型的事件侦听器，`hasEventListener()` 方法将返回 `true`。如果发现特定显示列表对象的侦听器，`willTrigger()` 方法也会返回 `true`。但 `willTrigger()` 不但检查该显示对象上的侦听器，还会检查该显示列表对象在事件流所有阶段中的所有父级上的侦听器。

## 没有侦听器的错误事件

ActionScript 3.0 中处理错误的主要机制是异常而不是事件，但对于异步操作（例如加载文件），异常处理不起作用。如果在这样的异步操作中发生错误， Flash Player 会调度一个错误事件对象。如果不为错误事件创建侦听器， Flash Player 的调试器版本将打开一个对话框，其中包含有关该错误的信息。例如，如果加载文件时使用无效 URL，将在 Flash Player 的调试器版本中生成此对话框：




大多数错误事件基于 ErrorEvent 类，而且同样具有一个名为 `text` 的属性，它用于存储 Flash Player 显示的错误消息。两个异常是 StatusEvent 和 NetStatusEvent 类。这两个类都具有一个 `level` 属性（`StatusEvent.level` 和 `NetStatusEvent.info.level`）。当 `level` 属性的值为 "`error`" 时，这些事件类型被视为错误事件。

错误事件将不会导致 SWF 文件停止运行。它仅在浏览器插件和独立播放器的调试器版本上显示为对话框，在创作播放器的输出面板中显示为消息，在 Adobe Flex Builder 2 的日志文件中显示为条目，而在 Flash Player 的发行版中则根本不显示。

# 示例： Alarm Clock

Alarm Clock 示例包含一个时钟，供用户指定闹铃响起的时间，还包含一个在该时间显示的消息。Alarm Clock 示例建立在 SimpleClock 应用程序的基础上，详见第 5 章 "处理日期和时间" Alarm Clock 阐明了使用 ActionScript 3.0 中的事件的一些方面，其中包括：

- 侦听和响应事件
- 向侦听器通知事件
- 创建自定义事件类型

要获取该范例的应用程序文件，请访问
www.adobe.com/go/learn_programmingAS3samples_flash_cn。可以在
Samples/AlarmClock 文件夹中找到 Alarm Clock 应用程序文件。该应用程序包括以下文件：

| 文件 | 说明 |
|---|---|
| AlarmClockApp.mxml<br>或<br>AlarmClockApp.fla | Flash 或 Flex 中的主应用程序文件 （分别为 FLA 和 MXML）。 |
| com/example/programmingas3/clock/AlarmClock.as | 一个扩展 SimpleClock 类的类，添加了闹钟功能。 |
| com/example/programmingas3/clock/AlarmEvent.as | 自定义事件类 （flash.events.Event 的子类），用作 AlarmClock 类的 `alarm` 事件的事件对象。 |
| com/example/programmingas3/clock/AnalogClockFace.as | 绘制一个圆的时钟形状以及基于时间的时针、分针和秒针 （如 SimpleClock 示例中所述）。 |
| com/example/programmingas3/clock/SimpleClock.as | 具有简单走时功能的时钟界面组件 （如 SimpleClock 示例中所述）。 |

# Alarm Clock 概述

在本示例中，时钟的主要功能 （包括跟踪时间和显示时钟形状）重复使用 SimpleClock 应用程序代码，详见第 166 页的 “示例：简单的模拟时钟”。AlarmClock 类添加了闹钟所需的功能 （包括设置闹铃时间和在闹铃响起时显示通知），从而扩展了该示例中的 SimpleClock 类。

在发生事情时提供通知，是创建事件的目的。AlarmClock 类公开 Alarm 事件，其它对象可侦听该事件以执行所需操作。此外，AlarmClock 类使用 Timer 类的实例来确定何时触发闹铃。和 AlarmClock 类一样，Timer 类提供一个事件，用于在经过特定时间时通知其它对象（在本例中为 AlarmClock 实例） 。就像大多数 ActionScript 应用程序一样，事件构成了 Alarm Clock 范例应用程序功能的重要部分。

# 触发闹铃

如前所述， AlarmClock 类实际提供的唯一功能与设置和触发闹铃有关。内置的 Timer 类 (flash.utils.Timer) 为开发人员提供了定义要在指定时间之后执行的代码的方法。AlarmClock 类使用 Timer 实例来确定何时触发闹铃。

```
import flash.events.TimerEvent;
import flash.utils.Timer;

/**
 * 将用于闹铃的 Timer。
 */
```

```
public var alarmTimer:Timer;
...
/**
 * 实例化指定大小的新 AlarmClock。
 */
public override function initClock(faceSize:Number = 200):void
{
  super.initClock(faceSize);
  alarmTimer = new Timer(0, 1);
  alarmTimer.addEventListener(TimerEvent.TIMER, onAlarm);
}
```

AlarmClock 类中定义的 Timer 实例被命名为 `alarmTimer`。`initClock()` 方法执行 AlarmClock 实例的所需设置操作，使用 `alarmTimer` 变量执行两个任务。首先，使用指示 Timer 实例等待 0 毫秒且仅触发其 timer 事件一次的参数实例化变量。实例化 `alarmTimer` 后，代码调用变量的 `addEventListener()` 方法，指示它要监听该变量的 `timer` 事件。Timer 实例的工作方式是：在经过指定时间后调度其 `timer` 事件。 AlarmClock 类需要了解何时调度 `timer` 事件，以便触发自己的闹铃。通过调用 `addEventListener()`， AlarmClock 代码将自身作为侦听器在 `alarmTimer` 中进行注册。两个参数指示 AlarmClock 类要侦听 `timer` 事件（由常量 `TimerEvent.TIMER` 指示），并且当事件发生时，应调用 AlarmClock 类的 `onAlarm()` 方法以响应事件。

为了实际设置闹铃，代码调用了 AlarmClock 类的 `setAlarm()` 方法，如下所示：

```
/**
 * 设置应触发闹铃的时间。
 * @param hour 闹铃时间的小时部分。
 * @param minutes 闹铃时间的分钟部分。
 * @param message 触发闹铃时显示的消息。
 * @return 将触发闹铃的时间。
 */
public function setAlarm(hour:Number = 0, minutes:Number = 0,
message:String = "Alarm!"):Date
{
  this.alarmMessage = message;
  var now:Date = new Date();
  // 将此时间作为今天的某个时间来创建。
  alarmTime = new Date(now.fullYear, now.month, now.date, hour, minutes);

  // 确定指定的时间是否在今天之后。
  if (alarmTime <= now)
  {
    alarmTime.setTime(alarmTime.time + MILLISECONDS_PER_DAY);
  }

  // 如果当前设置了闹铃计时器，则将其停止。
  alarmTimer.reset();
  // 计算过多少毫秒后，才会触发
  // 闹铃（闹铃时间和当前时间之差），并将该值
```

```
    // 设置为闹铃计时器的延时。
    alarmTimer.delay = Math.max(1000, alarmTime.time - now.time);
    alarmTimer.start();

    return alarmTime;
  }
```

此方法执行了几项操作，包括存储闹铃消息和创建一个 Date 对象 (alarmTime)，该对象表示触发闹铃的实际时间。在该方法的最后几行中，与当前讨论最相关的操作是设置和激活了 alarmTimer 变量的计时器。首先，调用其 reset() 方法，如果计时器已运行，则将其停止并进行重置。接下来，从 alarmTime 变量值中减去当前时间（由 now 变量表示），以确定需要经过多少毫秒后才会触发闹铃。 Timer 类并不会在某个绝对时间触发其 timer 事件，因此，分配给 alarmTimer 的 delay 属性的是该相对时间差。最后，调用 start() 方法以实际启动计时器。

一旦经过指定时间，alarmTimer 将调度 timer 事件。由于 AlarmClock 类已将其 onAlarm() 方法注册为该事件的侦听器，因此发生 timer 事件时，将调用 onAlarm()。

```
  /**
   * 调度 timer 事件时调用。
   */
  public function onAlarm(event:TimerEvent):void
  {
    trace("Alarm!");
    var alarm:AlarmEvent = new AlarmEvent(this.alarmMessage);
    this.dispatchEvent(alarm);
  }
```

注册为事件侦听器的方法必须使用适当的签名（即，方法的参数集和返回类型）来定义。要侦听 Timer 类的 timer 事件，方法必须定义一个数据类型为 TimerEvent (flash.events.TimerEvent) 的参数，该参数是 Event 类的子类。当 Timer 实例调用其事件侦听器时，会传递一个 TimerEvent 实例作为事件对象。

## 向其它代码通知闹铃

和 Timer 类一样， AlarmClock 类提供了一个事件，以允许其它代码在闹铃响起时收到通知。对于类而言，要使用内置于 ActionScript 中的事件处理框架，必须实现 flash.events.IEventDispatcher 接口。通常，这是通过扩展 flash.events.EventDispatcher 类（提供 IEventDispatcher 的标准实现）或 EventDispatcher 的某个子类来完成的。如前所述，AlarmClock 类扩展 SimpleClock 类，SimpleClock 类扩展 Sprite 类，而 Sprite 类（通过继承链）扩展 EventDispatcher 类。所有这些意味着 AlarmClock 类已经具有内置功能以提供自己的事件。

其它代码可通过调用 AlarmClock 从 EventDispatcher 继承的 `addEventListener()` 方法进行注册，以获得 AlarmClock 类的 `alarm` 事件的通知。当 AlarmClock 实例准备通知其它代码已引发其 `alarm` 事件时，它会调用 `dispatchEvent()` 方法进行通知，该方法同样是从 EventDispatcher 继承的。

```
var alarm:AlarmEvent = new AlarmEvent(this.alarmMessage);
this.dispatchEvent(alarm);
```

这些代码行摘自 AlarmClock 类的 `onAlarm()` 方法（前面完整介绍过）。调用 AlarmClock 实例的 `dispatchEvent()` 方法，该方法接下来通知所有注册的侦听器: 已触发 AlarmClock 实例的 `alarm` 事件。传递给 `dispatchEvent()` 的参数是要一直传递到侦听器方法的事件对象。在本例中，它是 AlarmEvent 类的实例，即为本示例专门创建的 Event 子类。

## 提供自定义 Alarm 事件

所有事件侦听器都接收一个事件对象参数，该参数提供有关要触发的特定事件的信息。在许多情况下，事件对象是 Event 类的实例。但在某些情况下，向事件侦听器提供其它信息很有用。如本章前面所述，实现该目的的一个常用方法是定义一个新类（Event 类的子类），并将该类的实例用作事件对象。在本示例中，当调度 AlarmClock 类的 `alarm` 事件时，会将一个 AlarmEvent 实例用作事件对象。在此介绍的 AlarmEvent 类提供有关 `alarm` 事件的其它信息，具体来说是闹铃消息:

```
import flash.events.Event;

/**
 * 此自定义 Event 类向基本 Event 添加一个 message 属性。
 */
public class AlarmEvent extends Event
{
  /**
   * 新 AlarmEvent 类型的名称。
   */
  public static const ALARM:String = "alarm";

  /**
   * 可以随该事件对象传递给事件处理函数
   * 的文本消息。
   */
  public var message:String;

  /**
   * 构造函数。
   * @param message 触发闹铃时显示的文本。
   */
  public function AlarmEvent(message:String = "ALARM!")
  {
    super(ALARM);
```

```
        this.message = message;
    }
    ...
}
```

要创建自定义事件对象类，最好的方法是定义一个扩展 Event 类的类，如前面的示例中所示。为了补充继承的功能， AlarmEvent 类定义一个属性 `message`，该属性包含与事件关联的闹铃消息的文本； `message` 是作为 AlarmEvent 构造函数中的参数传入的。AlarmEvent 类还定义常量 `ALARM`，当调用 AlarmClock 类的 `addEventListener()` 方法时，该常量可用于引用特定事件 (`alarm`)。

除了添加自定义功能外，作为 ActionScript 事件处理框架的一部分，每个 Event 子类还必须覆盖继承的 `clone()` 方法。Event 子类还可以选择性地覆盖继承的 `toString()` 方法，以便在调用 `toString()` 方法时返回的值中包括自定义事件的属性。

```
/**
 * 创建并返回当前实例的副本。
 * @return 当前实例的副本。
 */
public override function clone():Event
{
    return new AlarmEvent(message);
}

/**
 * 返回包含当前实例的所有属性的
 * 字符串。
 * @return 当前实例的字符串表示形式。
 */
public override function toString():String
{
    return formatToString("AlarmEvent", "type", "bubbles", "cancelable",
"eventPhase", "message");
}
```

被覆盖的 `clone()` 方法需要返回自定义 Event 子类的新实例，并且设置了所有自定义属性以匹配当前实例。在被覆盖的 `toString()` 方法中，实用程序方法 `formatToString()` （从 Event 继承）用于提供一个字符串，包括自定义类型的名称以及所有属性的名称和值。

第 11 章

# 处理 XML

11

ActionScript 3.0 包含一组基于 ECMAScript for XML (E4X) 规范（ECMA-357 第 2 版）的类。这些类包含用于处理 XML 数据的强大且易用的功能。与以前的编程技术相比，使用 E4X 可以更快地用 XML 数据开发代码。此外，开发的代码也更容易阅读。

本章介绍如何使用 E4X 来处理 XML 数据。

## 目录

# XML 基础知识

## 处理 XML 简介

XML 是一种表示结构化信息的标准方法，以使计算机能够方便地使用此类信息，并且人们可以非常方便地编写和理解这些信息。XML 是 eXtensible Markup Language（可扩展标记语言）的缩写。 www.w3.org/XML/ 上提供了 XML 标准。

XML 提供了一种简便的标准方法对数据进行分类，以使其更易于读取、访问以及处理。XML 使用类似于 HTML 的树结构和标签结构。以下是一个简单的 XML 数据示例：

```
<song>
  <title>What you know?</title>
  <artist>Steve and the flubberblubs</artist>
  <year>1989</year>
  <lastplayed>2006-10-17-08:31</lastplayed>
</song>
```

XML 数据也可能会比较复杂，其中包含嵌套在其它标签中的标签以及属性和其它结构组件。以下是一个比较复杂的 XML 数据示例：

```
<album>
  <title>Questions, unanswered</title>
  <artist>Steve and the flubberblubs</artist>
  <year>1989</year>
  <tracks>
    <song tracknumber="1" length="4:05">
      <title>What do you know?</title>
      <artist>Steve and the flubberblubs</artist>
      <lastplayed>2006-10-17-08:31</lastplayed>
    </song>
    <song tracknumber="2" length="3:45">
      <title>Who do you know?</title>
      <artist>Steve and the flubberblubs</artist>
      <lastplayed>2006-10-17-08:35</lastplayed>
    </song>
    <song tracknumber="3" length="5:14">
      <title>When do you know?</title>
      <artist>Steve and the flubberblubs</artist>
      <lastplayed>2006-10-17-08:39</lastplayed>
    </song>
    <song tracknumber="4" length="4:19">
      <title>Do you know?</title>
      <artist>Steve and the flubberblubs</artist>
      <lastplayed>2006-10-17-08:44</lastplayed>
    </song>
  </tracks>
</album>
```

请注意，此 XML 文档中包含其它完整 XML 结构 （如 song 标签及其子标签）。它还说明了其它 XML 结构，如属性 （song 标签中的 tracknumber 和 length）以及包含其它标签而不是包含数据的标签 （如 tracks 标签）。

## XML 快速入门

如果您没有或几乎没有 XML 方面的经验，您可以阅读下面对 XML 数据的最常见特性的简要说明。 XML 数据是以纯文本格式编写的，并使用特定语法将信息组织为结构化格式。通常，将一组 XML 数据称为 “XML 文档”。在 XML 格式中，可通过分层结构将数据组织到 “元素”（可以是单个数据项，也可以是其它元素的容器）中。每个 XML 文档将一个元素作为顶级项目或主项目；此根元素内可能会包含一条信息，但更可能会包含其它元素，而这些元素又包含其它元素，依此类推。例如，以下 XML 文档包含有关音乐唱片的信息：

```
<song tracknumber="1" length="4:05">
  <title>What do you know?</title>
  <artist>Steve and the flubberblubs</artist>
  <mood>Happy</mood>
  <lastplayed>2006-10-17-08:31</lastplayed>
</song>
```

每个元素都是用一组 “标签”来区分的，即包含在尖括号 （小于号和大于号）中的元素名称。开始标签 （指示元素的开头）包含元素名称：

```
<title>
```

结束标签 （标记元素的结尾）在元素名称前面包含一个正斜杠：

```
</title>
```

如果元素不包含任何内容，则会将其编写为一个空元素 （有时称为自结束元素）。在 XML 中，以下元素：

```
<lastplayed/>
```

与下面的元素完全相同：

```
<lastplayed></lastplayed>
```

除了在开始和结束标签之间包含的元素内容外，元素还可以包含在元素开始标签中定义的其它值（称为“属性”）。例如，以下 XML 元素定义一个名为 length 且值为“4:19”的属性：

```
<song length="4:19"></song>
```

每个 XML 元素都包含内容，这可以是单个值、一个或多个 XML 元素或没有任何内容（对于空元素）。

## 了解有关 XML 的详细信息

要了解有关处理 XML 的详细信息，请参阅额外的一些书籍和资源以了解有关 XML 的详细信息，其中包括以下 Web 站点：

- W3Schools XML 教程：http://w3schools.com/xml/
- XML.com：http://www.xml.com/
- XMLpitstop 教程、讨论列表等等：http://xmlpitstop.com/

## 用于处理 XML 的 ActionScript 类

ActionScript 3.0 包含一些用于处理 XML 结构化信息的类。下面列出了两个主类：

- XML：表示单个 XML 元素，它可以是包含多个子元素的 XML 文档，也可以是文档中的单值元素。
- XMLList：表示一组 XML 元素。当具有多个 "同级"（在 XML 文档分层结构中的相同级别，并且包含在相同父级中）的 XML 元素时，将使用 XMLList 对象。例如，XMLList 实例是使用以下一组 XML 元素（可能包含在 XML 文档中）的最简便方法：

```
<artist type="composer">Fred Wilson</artist>
<artist type="conductor">James Schmidt</artist>
<artist type="soloist">Susan Harriet Thurndon</artist>
```

对于涉及 XML 命名空间的更高级用法，ActionScript 还包含 Namespace 和 QName 类。有关详细信息，请参阅第 311 页的 "使用 XML 命名空间"。

除了用于处理 XML 的内置类外，ActionScript 3.0 还包含一些运算符，它们提供了用于访问和处理 XML 数据的特定功能。这种使用这些类和运算符来处理 XML 的方法称为 ECMAScript for XML (E4X)，它是由 ECMA-357 第 2 版规范定义的。

# 常见 XML 任务

在 ActionScript 中处理 XML 时，您可能会执行以下任务：

- 构造 XML 文档（添加元素和值）
- 访问 XML 元素、值和属性
- 过滤（从中搜索）XML 元素
- 循环访问一组 XML 元素
- 在 XML 类和 String 类之间转换数据
- 处理 XML 命名空间
- 加载外部 XML 文件

## 重要概念和术语

以下参考列表包含将会在本章中使用的重要术语：

- 元素 (Element)：XML 文档中的单个项目，它被标识为开始标签和结束标签之间包含的内容（包括标签）。XML 元素可以包含文本数据或其它元素，也可以为空。
- 空元素 (Empty element)：不包含任何子元素的 XML 元素。通常，将空元素编写为自结束标签（如 `<element/>`）。
- 文档 (Document)：单个 XML 结构。XML 文档可以包含任意数量的元素（或者仅包含单个空元素）；但是，XML 文档必须具有一个顶级元素，该元素包含文档中的所有其它元素。
- 节点 (Node)：XML 元素的另一种称谓。
- 属性 (Attribute)：与元素关联的命名值，它以 `attributename="value"` 格式写入到元素的开始标签中，而不是编写为嵌套在元素内的单独子元素。

## 完成本章中的示例

学习本章的过程中，您可能想要自己动手测试一些示例代码清单。实质上本章中的代码清单已经包括适当的 `trace()` 函数调用。要测试本章中的代码清单，请执行以下操作：

1. 创建一个空的 Flash 文档。
2. 在时间轴上选择一个关键帧。
3. 打开“动作”面板，将代码清单复制到“脚本”窗格中。
4. 使用“控制”>“测试影片”运行程序。

   可在“输出”面板中看到 `trace()` 函数的结果。

第 53 页的“测试本章内的示例代码清单”中对用于测试代码清单的此技术和其它技术进行了详细说明。

# 用于处理 XML 的 E4X 方法

ECMAScript for XML 规范定义了一组用于处理 XML 数据的类和功能。这些类和功能统称为 *E4X*。ActionScript 3.0 包含以下 E4X 类：XML、XMLList、QName 和 Namespace。

E4X 类的方法、属性和运算符旨在实现以下目标：

- 简单 — 在可能的情况下，使用 E4X 可以更容易地编写和理解用于处理 XML 数据的代码。
- 一致 — E4X 背后的方法和推理在内部是一致的，并与 ActionScript 的其它部分保持一致。

■ 熟悉 — 使用众所周知的运算符来处理 XML 数据，如点 (.) 运算符。

> **注意** ActionScript 2.0 中有一个 XML 类。在 ActionScript 3.0 中，此类已重新命名为 XMLDocument，从而避免与作为 E4X 的一部分的 ActionScript 3.0 XML 类冲突。在 ActionScript 3.0 中，flash.xml 包中包含了 XMLDocument、XMLNode、XMLParser 和 XMLTag 几个旧类，主要是用于旧支持。新的 E4X 类是核心类；无需导入包即可使用这些类。本章不详细阐述旧的 ActionScript 2.0 XML 类。有关这些类的详细信息，请参阅《ActionScript 3.0 语言和组件参考》中的 flash.xml 包。

下面是使用 E4X 处理数据的一个示例：

```
var myXML:XML =
  <order>
    <item id='1'>
      <menuName>burger</menuName>
      <price>3.95</price>
    </item>
    <item id='2'>
      <menuName>fries</menuName>
      <price>1.45</price>
    </item>
  </order>
```

通常，应用程序都会从外部源（如 Web 服务或 RSS 供给）加载 XML 数据。不过为清楚起见，本章中的示例是将 XML 数据作为文本来分配的。

如以下代码所示，E4X 包含了一些直观运算符（如点 (.)）和属性标识符 (@) 运算符，用于访问 XML 中的属性 (property) 和属性 (attribute)：

```
trace(myXML.item[0].menuName); // 输出: burger
trace(myXML.item.(@id==2).menuName); // 输出: fries
trace(myXML.item.(menuName=="burger").price); // 输出: 3.95
```

使用 `appendChild()` 方法可为 XML 分配一个新的子节点，如下面的代码片断所示：

```
var newItem:XML =
  <item id="3">
    <menuName>medium cola</menuName>
    <price>1.25</price>
  </item>

myXML.appendChild(newItem);
```

使用 @ 和 . 运算符不仅可以读取数据，还可以分配数据，如下所示：

```
myXML.item[0].menuName="regular burger";
myXML.item[1].menuName="small fries";
myXML.item[2].menuName="medium cola";

myXML.item.(menuName=="regular burger").@quantity = "2";
myXML.item.(menuName=="small fries").@quantity = "2";
myXML.item.(menuName=="medium cola").@quantity = "2";
```

使用 `for` 循环可以循环访问 XML 的节点，如下所示：

```
var total:Number = 0;
for each (var property:XML in myXML.item)
{
  var q:int = Number(property.@quantity);
  var p:Number = Number(property.price);
  var itemTotal:Number = q * p;
  total += itemTotal;
  trace(q + " " + property.menuName + " $" + itemTotal.toFixed(2))
}
trace("Total: $", total.toFixed(2));
```

# XML 对象

XML 对象可能表示 XML 元素、属性、注释、处理指令或文本元素。

XML 对象分为包含“简单内容”和包含“复杂内容”两类。有子节点的 XML 对象归入包含复杂内容的一类。如果 XML 对象是属性、注释、处理指令或文本节点之中的任何一个，我们就说它包含简单内容。

例如，下面的 XML 对象包含复杂内容，包括一条注释和一条处理指令：

```
XML.ignoreComments = false;
XML.ignoreProcessingInstructions = false;
var x1:XML =
  <order>
    <!-- 这是一条注释。 -->
    <?PROC_INSTR sample ?>
    <item id='1'>
      <menuName>burger</menuName>
      <price>3.95</price>
    </item>
    <item id='2'>
      <menuName>fries</menuName>
      <price>1.45</price>
    </item>
  </order>
```

如下面的示例所示，现在可以使用 `comments()` 和 `processingInstructions()` 方法创建新的 XML 对象，即注释和处理指令：

```
var x2:XML = x1.comments()[0];
var x3:XML = x1.processingInstructions()[0];
```

## XML 属性

XML 类有五个静态属性：

- `ignoreComments` 和 `ignoreProcessingInstructions` 属性确定分析 XML 对象时是否忽略注释或处理指令。
- `ignoreWhitespace` 属性确定在只由空白字符分隔的元素标签和内嵌表达式中是否忽略空白字符。
- `prettyIndent` 和 `prettyPrinting` 属性用于设置由 XML 类的 `toString()` 和 `toXMLString()` 方法返回的文本的格式。

有关这些属性的详细信息，请参阅 《ActionScript 3.0 语言和组件参考》。

## XML 方法

以下方法用于处理 XML 对象的分层结构：

- `appendChild()`
- `child()`
- `childIndex()`
- `children()`
- `descendants()`
- `elements()`
- `insertChildAfter()`
- `insertChildBefore()`
- `parent()`
- `prependChild()`

以下方法用于处理 XML 对象属性 (attribute)：

- `attribute()`
- `attributes()`

以下方法用于处理 XML 对象属性 (property)：

- `hasOwnProperty()`
- `propertyIsEnumerable()`
- `replace()`
- `setChildren()`

以下方法用于处理限定名和命名空间:

- addNamespace()
- inScopeNamespaces()
- localName()
- name()
- namespace()
- namespaceDeclarations()
- removeNamespace()
- setLocalName()
- setName()
- setNamespace()

以下方法用于处理和确定某些类型的 XML 内容:

- comments()
- hasComplexContent()
- hasSimpleContent()
- nodeKind()
- processingInstructions()
- text()

以下方法用于转换为字符串和设置 XML 对象的格式:

- defaultSettings()
- setSettings()
- settings()
- normalize()
- toString()
- toXMLString()

还有其它几个方法:

- contains()
- copy()
- valueOf()
- length()

有关这些方法的详细信息,请参阅 《ActionScript 3.0 语言和组件参考》。

# XMLList 对象

XMLList 实例表示 XML 对象的任意集合。它可以包含完整的 XML 文档、XML 片断或 XML 查询结果。

以下方法用于处理 XMLList 对象的分层结构:

- `child()`
- `children()`
- `descendants()`
- `elements()`
- `parent()`

以下方法用于处理 XMLList 对象属性 (attribute):

- `attribute()`
- `attributes()`

以下方法用于处理 XMLList 属性 (property):

- `hasOwnProperty()`
- `propertyIsEnumerable()`

以下方法用于处理和确定某些类型的 XML 内容:

- `comments()`
- `hasComplexContent()`
- `hasSimpleContent()`
- `processingInstructions()`
- `text()`

以下方法用于转换为字符串和设置 XMLList 对象的格式:

- `normalize()`
- `toString()`
- `toXMLString()`

还有其它几个方法:

- `contains()`
- `copy()`
- `length()`
- `valueOf()`

有关这些方法的详细信息，请参阅 《ActionScript 3.0 语言和组件参考》。

对于只包含一个 XML 元素的 XMLList 对象，可以使用 XML 类的所有属性和方法，因为包含一个 XML 元素的 XMLList 被视为等同于 XML 对象。例如，在下面的代码中，因为 doc.div 是包含一个元素的 XMLList 对象，所以可以使用 XML 类的 appendChild() 方法：

```
var doc:XML =
    <body>
      <div>
        <p>Hello</p>
      </div>
    </body>;
doc.div.appendChild(<p>World</p>);
```

有关 XML 属性和方法的列表，请参阅第 299 页的 “XML 对象”。

# 初始化 XML 变量

可将 XML 文本赋予 XML 对象，如下所示：

```
var myXML:XML =
  <order>
    <item id='1'>
      <menuName>burger</menuName>
      <price>3.95</price>
    </item>
    <item id='2'>
      <menuName>fries</menuName>
      <price>1.45</price>
    </item>
  </order>
```

如下面的代码片断所示，还可以使用 new 构造函数从包含 XML 数据的字符串创建 XML 对象的实例：

```
var str:String = "<order><item id='1'><menuName>burger</menuName>"
                + "<price>3.95</price></item></order>";
var myXML:XML = new XML(str);
```

如果字符串中的 XML 数据格式有误 （例如缺少结束标签），则会出现运行时错误。

还可以将数据按引用 （从其它变量）传递到 XML 对象，如下面的示例所示：

```
var tagname:String = "item";
var attributename:String = "id";
var attributevalue:String = “5”;
var content:String = "Chicken";
var x:XML = <{tagname} {attributename}={attributevalue}>{content}</
  {tagname}>;
trace(x.toXMLString())
  // 输出：<item id="5">Chicken</item>
```

要从 URL 加载 XML 数据，请使用 URLLoader 类，如下面的示例所示：

```
import flash.events.Event;
import flash.net.URLLoader;
import flash.net.URLRequest;

var externalXML:XML;
var loader:URLLoader = new URLLoader();
var request:URLRequest = new URLRequest("xmlFile.xml");
loader.load(request);
loader.addEventListener(Event.COMPLETE, onComplete);

function onComplete(event:Event):void
{
  var loader:URLLoader = event.target as URLLoader;
  if (loader != null)
  {
    externalXML = new XML(loader.data);
    trace(externalXML.toXMLString());
  }
  else
  {
    trace("loader is not a URLLoader!");
  }
}
```

要从套接字连接读取 XML 数据，请使用 XMLSocket 类。有关详细信息，请参阅《ActionScript 3.0 语言和组件参考》中的 XMLSocket 条目。

# 组合和变换 XML 对象

使用 `prependChild()` 方法或 `appendChild()` 方法可在 XML 对象属性列表的开头或结尾添加属性，如下面的示例所示：

```
var x1:XML = <p>Line 1</p>
var x2:XML = <p>Line 2</p>
var x:XML = <body></body>
x = x.appendChild(x1);
x = x.appendChild(x2);
x = x.prependChild(<p>Line 0</p>);
  // x == <body><p>Line 0</p><p>Line 1</p><p>Line 2</p></body>
```

使用 insertChildBefore() 方法或 insertChildAfter() 方法在指定属性之前或之后添加属性，如下所示：

```
var x:XML =
  <body>
    <p>Paragraph 1</p>
    <p>Paragraph 2</p>
  </body>
var newNode:XML = <p>Paragraph 1.5</p>
x = x.insertChildAfter(x.p[0], newNode)
x = x.insertChildBefore(x.p[2], <p>Paragraph 1.75</p>)
```

如下面的示例所示，还可以使用大括号运算符（{ 和 }）在构造 XML 对象时按引用（从其它变量）传递数据：

```
var ids:Array = [121, 122, 123];
var names:Array = [["Murphy","Pat"], ["Thibaut","Jean"], ["Smith","Vijay"]]
var x:XML = new XML("<employeeList></employeeList>");

for (var i:int = 0; i < 3; i++)
{
  var newnode:XML = new XML();
  newnode =
    <employee id={ids[i]}>
      <last>{names[i][0]}</last>
      <first>{names[i][1]}</first>
    </employee>;

  x = x.appendChild(newnode)
}
```

可以使用 = 运算符将属性 (property) 和属性 (attribute) 赋予 XML 对象，如下所示：

```
var x:XML =
  <employee>
    <lastname>Smith</lastname>
  </employee>
x.firstname = "Jean";
x.@id = "239";
```

这将对 XML 对象 x 进行如下设置：

```
<employee id="239">
  <lastname>Smith</lastname>
  <firstname>Jean</firstname>
</employee>
```

可以使用 + 和 += 运算符连接 XMLList 对象：

```
var x1:XML = <a>test1</a>
var x2:XML = <b>test2</b>
var xList:XMLList = x1 + x2;
xList += <c>test3</c>
```

这将对 XMLList 对象 `xList` 进行如下设置：

```
<a>test1</a>
<b>test2</b>
<c>test3</c>
```

# 遍历 XML 结构

XML 的一个强大功能是它能够通过文本字符的线性字符串提供复杂的嵌套数据。将数据加载到 XML 对象时， ActionScript 会分析数据并将其分层结构加载到内存 （如果 XML 数据格式有误，它会发送运行时错误）。

利用 XML 和 XMLList 对象的运算符和方法可以轻松遍历 XML 数据的结构。

使用点 (.) 运算符和后代存取器 (..) 运算符可以访问 XML 对象的子属性。请考虑下面的 XML 对象：

```
var myXML:XML =
  <order>
    <book ISBN="0942407296">
      <title>Baking Extravagant Pastries with Kumquats</title>
      <author>
        <lastName>Contino</lastName>
        <firstName>Chuck</firstName>
      </author>
      <pageCount>238</pageCount>
    </book>
    <book ISBN="0865436401">
      <title>Emu Care and Breeding</title>
      <editor>
        <lastName>Case</lastName>
        <firstName>Justin</firstName>
      </editor>
      <pageCount>115</pageCount>
    </book>
  </order>
```

对象 `myXML.book` 是一个 XMLList 对象，它包含名为 `book` 的 `myXML` 对象的子属性。它们是两个 XML 对象，与 `myXML` 对象的两个 `book` 属性匹配。

对象 `myXML..lastName` 是一个 XMLList 对象，它包含名为 `lastName` 的任何后代属性。它们是两个 XML 对象，与 `myXML` 对象的两个 `lastName` 匹配。

`myXML.book.editor.lastName` 对象是一个 XMLList 对象，它包含 `myXML` 对象的名为 `book` 的子对象的名为 `editor` 的子对象的名为 `lastName` 的任何子对象：在本例中，XMLList 对象只包含一个 XML 对象 （值为 “`Case`”的 `lastName` 属性）。

## 访问父节点和子节点

parent() 方法返回 XML 对象的父项。

可以使用子级列表的序数索引值访问特定的子对象。例如，假设 XML 对象 myXML 有两个名为 book 的子属性。每个名为 book 的子属性都有一个与之关联的索引编号：

```
myXML.book[0]
myXML.book[1]
```

要访问特定的孙项，可为子项和孙项名称同时指定索引编号：

```
myXML.book[0].title[0]
```

不过，如果 x.book[0] 只有一个子项名为 title，则可以省略索引引用，如下所示：

```
myXML.book[0].title
```

同样，如果对象 x 只有一个 book 子对象，并且如果该子对象只有一个 title 对象，则可以同时省略两个索引引用，如下所示：

```
myXML.book.title
```

可以使用 child() 方法导航到名称基于变量或表达式的子项，如下面的示例所示：

```
var myXML:XML =
     <order>
        <book>
          <title>Dictionary</title>
        </book>
     </order>;

var childName:String = "book";

trace(myXML.child(childName).title) // 输出: Dictionary
```

## 访问属性

使用 @ 符号（属性标识符运算符）可以访问 XML 或 XMLList 对象的属性，如下面的代码所示：

```
var employee:XML =
  <employee id="6401" code="233">
     <lastName>Wu</lastName>
     <firstName>Erin</firstName>
  </employee>;
trace(employee.@id); // 6401
```

可以一起使用 * 通配符和 @ 符号来访问 XML 或 XMLList 对象的所有属性，如下面的代码所示：

```
var employee:XML =
  <employee id="6401" code="233">
    <lastName>Wu</lastName>
    <firstName>Erin</firstName>
  </employee>;
trace(employee.@*.toXMLString());
// 6401
// 233
```

可以使用 `attribute()` 或 `attributes()` 方法访问 XML 或 XMLList 对象的特定属性或所有属性，如下面的代码所示：

```
var employee:XML =
  <employee id="6401" code="233">
    <lastName>Wu</lastName>
    <firstName>Erin</firstName>
  </employee>;
trace(employee.attribute("id")); // 6401
trace(employee.attribute("*").toXMLString());
// 6401
// 233
trace(employee.attributes().toXMLString());
// 6401
// 233
```

请注意，还可以使用以下语法访问属性，如下面的示例所示：

```
employee.attribute("id")
employee["@id"]
employee.@["id"]
```

其中每一个都等效于 `employee.@id`。不过，语法 `employee.@id` 是首选方法。

## 按属性或元素值过滤

可以使用括号运算符 —（ 和 ）— 过滤具有特定元素名称或属性值的元素。请考虑下面的 XML 对象：

```
var x:XML =
  <employeeList>
    <employee id="347">
      <lastName>Zmed</lastName>
      <firstName>Sue</firstName>
      <position>Data analyst</position>
    </employee>
    <employee id="348">
      <lastName>McGee</lastName>
      <firstName>Chuck</firstName>
      <position>Jr. data analyst</position>
    </employee>
  </employeeList>
```

以下表达式都是有效的：

- `x.employee.(lastName == "McGee")` — 这是第二个 `employee` 节点。
- `x.employee.(lastName == "McGee").firstName` — 这是第二个 `employee` 节点的 `firstName` 属性。
- `x.employee.(lastName == "McGee").@id` — 这是第二个 `employee` 节点的 `id` 属性的值。
- `x.employee.(@id == 347)` — 第一个 `employee` 节点。
- `x.employee.(@id == 347).lastName` — 这是第一个 `employee` 节点的 `lastName` 属性。
- `x.employee.(@id > 300)` — 这是具有两个 `employee` 属性的 XMLList。
- `x.employee.(position.toString().search("analyst") > -1)` — 这是具有两个 `position` 属性的 XMLList。

如果试图按照可能不存在的属性或元素过滤，Adobe Flash Player 将引发异常。例如，以下代码的最后一行产生一个错误，因为第二个 `p` 元素中没有 `id` 属性：

```
var doc:XML =
       <body>
          <p id='123'>Hello, <b>Bob</b>.</p>
          <p>Hello.</p>
       </body>;
trace(doc.p.(@id == '123'));
```

同样，以下代码的最后一行也会产生一个错误，因为第二个 `p` 元素没有 `b` 属性：

```
var doc:XML =
       <body>
          <p id='123'>Hello, <b>Bob</b>.</p>
          <p>Hello.</p>
       </body>;
trace(doc.p.(b == 'Bob'));
```

为了避免这些错误，可以使用 `attribute()` 和 `elements()` 方法来识别具有匹配属性或元素的属性，如下面的代码所示：

```
var doc:XML =
       <body>
          <p id='123'>Hello, <b>Bob</b>.</p>
          <p>Hello.</p>
       </body>;
trace(doc.p.(attribute('id') == '123'));
trace(doc.p.(elements('b') == 'Bob'));
```

还可以使用 `hasOwnProperty()` 方法，如下面的代码所示：

```
var doc:XML =
       <body>
         <p id='123'>Hello, <b>Bob</b>.</p>
         <p>Hello.</p>
       </body>;
trace(doc.p.(hasOwnProperty('@id') && @id == '123'));
trace(doc.p.(hasOwnProperty('b') && b == 'Bob'));
```

## 使用 for..in 和 for each..in 语句

ActionScript 3.0 包含用于循环访问 XMLList 对象的 `for..in` 语句和 `for each..in` 语句。例如，我们来看 XML 对象 `myXML` 和 XMLList 对象 `myXML.item`。XMLList 对象 `myXML.item` 由 XML 对象的两个 `item` 节点组成。

```
var myXML:XML =
  <order>
    <item id='1' quantity='2'>
      <menuName>burger</menuName>
      <price>3.95</price>
    </item>
    <item id='2' quantity='2'>
      <menuName>fries</menuName>
      <price>1.45</price>
    </item>
  </order>;
```

`for..in` 语句用于循环访问 XMLList 中的一组属性名称：

```
var total:Number = 0;
for (var pname:String in myXML.item)
{
  total += myXML.item.@quantity[pname] * myXML.item.price[pname];
}
```

`for each..in` 语句用于循环访问 XMLList 中的属性：

```
var total2:Number = 0;
for each (var prop:XML in myXML.item)
{
  total2 += prop.@quantity * prop.price;
}
```

# 使用 XML 命名空间

XML 对象 （或文档）中的命名空间用于标识对象所包含的数据的类型。例如，在将 XML 数据发送和提交给使用 SOAP 消息传递协议的 Web 服务时，您要在 XML 的开始标签中声明命名空间：

```
var message:XML =
  <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <soap:Body xmlns:w="http://www.test.com/weather/">
      <w:getWeatherResponse>
        <w:tempurature >78</w:tempurature>
      </w:getWeatherResponse>
    </soap:Body>
  </soap:Envelope>;
```

命名空间有一个前缀 soap 和一个定义该命名空间的 URI http://schemas.xmlsoap.org/soap/envelope/。

ActionScript 3.0 包含用于处理 XML 命名空间的命名空间类。对于上一示例中的 XML 对象，可按如下方式使用命名空间类：

```
var soapNS:Namespace = message.namespace("soap");
trace(soapNS); // 输出: http://schemas.xmlsoap.org/soap/envelope/

var wNS:Namespace = new Namespace("w", "http://www.test.com/weather/");
message.addNamespace(wNS);
var encodingStyle:XMLList = message.@soapNS::encodingStyle;
var body:XMLList = message.soapNS::Body;

message.soapNS::Body.wNS::GetWeatherResponse.wNS::tempurature = "78";
```

XML 类包含用于处理命名空间的以下方法：addNamespace()、inScopeNamespaces()、localName()、name()、namespace()、namespaceDeclarations()、removeNamespace()、setLocalName()、setName() 和 setNamespace()。

default xml namespace 指令用于为 XML 对象指定默认的命名空间。例如，在以下代码中，x1 和 x2 具有相同的默认命名空间：

```
var ns1:Namespace = new Namespace("http://www.example.com/namespaces/");
default xml namespace = ns1;
var x1:XML = <test1 />;
var x2:XML = <test2 />;
```

# XML 类型转换

可以将 XML 对象和 XMLList 对象转换为字符串值。同样，也可以将字符串转换为 XML 对象和 XMLList 对象。还要记住，所有 XML 属性值、名称和文本值都是字符串。以下几节将讨论所有这些形式的 XML 类型转换。

## 将 XML 和 XMLList 对象转换为字符串

XML 和 XMLList 类都包含一个 `toString()` 方法和一个 `toXMLString()` 方法。`toXMLString()` 方法返回包含该 XML 对象的所有标签、属性、命名空间声明和内容的字符串。对于包含复杂内容（子元素）的 XML 对象，`toString()` 方法的作用与 `toXMLString()` 方法完全相同。对于包含简单内容的 XML 对象（只包含一个文本元素的对象），`toString()` 方法只返回该元素的文本内容，如下面的示例所示：

```
var myXML:XML =
  <order>
    <item id='1' quantity='2'>
      <menuName>burger</menuName>
      <price>3.95</price>
    </item>
  <order>;

trace(myXML.item[0].menuName.toXMLString());
  // <menuName>burger</menuName>
trace(myXML.item[0].menuName.toString());
  // burger
```

如果使用 `trace()` 方法但不指定 `toString()` 或 `toXMLString()`，则默认情况下将使用 `toString()` 方法转换数据，如以下代码所示：

```
var myXML:XML =
  <order>
    <item id='1' quantity='2'>
      <menuName>burger</menuName>
      <price>3.95</price>
    </item>
  <order>;

trace(myXML.item[0].menuName);
  // burger
```

使用 `trace()` 方法调试代码时，通常都要使用 `toXMLString()` 方法，以便 `trace()` 方法输出更完整的数据。

## 将字符串转换为 XML 对象

可以使用 `new XML()` 构造函数从字符串创建 XML 对象，如下所示：

```
var x:XML = new XML("<a>test</a>");
```

如果试图将表示无效 XML 或格式有误的 XML 的字符串转换为 XML，则会引发运行时错误，如下所示：

```
var x:XML = new XML("<a>test"); // 引发错误
```

## 从字符串转换属性值、名称和文本值

所有 XML 属性值、名称和文本值都是 String 数据类型，可能需要将它们转换为其它数据类型。例如，以下代码使用 `Number()` 函数将文本值转换为数字：

```
var myXML:XML = 
            <order>
              <item>
                <price>3.95</price>
              </item>
              <item>
                <price>1.00</price>
              </item>
            </order>;

var total:XML = <total>0</total>;
myXML.appendChild(total);

for each (var item:XML in myXML.item)
{
  myXML.total.children()[0] = Number(myXML.total.children()[0])
                        + Number(item.price.children()[0]);
}
trace(myXML.total); // 4.35;
```

如果这些代码不使用 `Number()` 函数，它们将把 + 运算符解释为字符串连接运算符，并且最后一行中的 `trace()` 方法将输出以下结果：

```
01.003.95
```

# 读取外部 XML 文档

可以使用 URLLoader 类从 URL 加载 XML 数据。要在应用程序中使用以下代码，请将示例中的 XML_URL 值替换为有效的 URL：

```
var myXML:XML = new XML();
var XML_URL:String = "http://www.example.com/Sample3.xml";
var myXMLURL:URLRequest = new URLRequest(XML_URL);
var myLoader:URLLoader = new URLLoader(myXMLURL);
myLoader.addEventListener("complete", xmlLoaded);

function xmlLoaded(event:Event):void
{
  myXML = XML(myLoader.data);
  trace("Data loaded.");
}
```

还可以使用 XMLSocket 类设置与服务器的异步 XML 套接字连接。有关详细信息，请参阅《ActionScript 3.0 语言和组件参考》。

# 示例：从 Internet 加载 RSS 数据

RSSViewer 范例应用程序说明了在 ActionScript 中处理 XML 的一些功能，其中包括：

- 使用 XML 方法以 RSS 供给的形式遍历 XML 数据。
- 使用 XML 方法以 HTML 的形式组合 XML 数据，以便在文本字段中使用。

RSS 格式被广泛用于通过 XML 收集新闻。简单的 RSS 数据文件可能如下所示：

```
<?xml version="1.0" encoding="UTF-8" ?>
<rss version="2.0" xmlns:dc="http://purl.org/dc/elements/1.1/">
<channel>
  <title>Alaska - Weather</title>
  <link>http://www.nws.noaa.gov/alerts/ak.html</link>
  <description>Alaska - Watches, Warnings and Advisories</description>

  <item>
    <title>
      Short Term Forecast - Taiya Inlet, Klondike Highway (Alaska)
    </title>
    <link>
      http://www.nws.noaa.gov/alerts/ak.html#A18.AJKNK.1900
    </link>
    <description>
      Short Term Forecast Issued At: 2005-04-11T19:00:00
      Expired At: 2005-04-12T01:00:00 Issuing Weather Forecast Office
      Homepage: http://pajk.arh.noaa.gov
    </description>
  </item>
  <item>
```

```
    <title>
      Short Term Forecast - Haines Borough (Alaska)
    </title>
    <link>
      http://www.nws.noaa.gov/alerts/ak.html#AKZ019.AJKNOWAJK.190000
    </link>
    <description>
      Short Term Forecast Issued At: 2005-04-11T19:00:00
      Expired At: 2005-04-12T01:00:00 Issuing Weather Forecast Office
      Homepage: http://pajk.arh.noaa.gov
    </description>
  </item>
</channel>
</rss>
```

SimpleRSS 应用程序从 Internet 上读取 RSS 数据，分析标题、链接和描述的数据，并返回这些数据。SimpleRSSUI 类提供了相应的用户界面，并会调用 SimpleRSS 类，从而执行所有 XML 处理。

要获取该范例的应用程序文件，请访问 www.adobe.com/go/learn_programmingAS3samples_flash_cn。RSSViewer 应用程序文件位于 Samples/RSSViewer 文件夹中。该应用程序包含以下文件：

| 文件 | 说明 |
|---|---|
| RSSViewer.mxml<br>或<br>RSSViewer.fla | Flash 或 Flex 中的主应用程序文件 （分别为 FLA 和 MXML）。 |
| com/example/programmingas3/rssViewer/RSSParser.as | 一个类，包含的方法可以使用 E4X 遍历 RSS (XML) 数据并生成相应的 HTML 表示形式。 |
| RSSData/ak.rss | 一个 RSS 范例文件。该应用程序被设置为从 Web 上由 Adobe 托管的 Flex RSS 供给处读取 RSS 数据。不过，您也可以轻松更改该应用程序，使之从此文档读取 RSS 数据，此文档所用的架构与 Flex RSS 供给的架构略有不同。 |

# 读取和分析 XML 数据

RSSParser 类包含一个 `xmlLoaded()` 方法，该方法可将输入 RSS 数据 （存储在 `rssXML` 变量中）转换为包含 HTML 格式的输出 (`rssOutput`) 的字符串。

如果源 RSS 数据包含默认的命名空间，代码会在此方法的开头附近设置默认的 XML 命名空间：

```
if (rssXML.namespace("") != undefined)
{
  default xml namespace = rssXML.namespace("");
}
```

下面的几行代码循环访问源 XML 数据的内容，以检查名为 item 的各个后代属性：

```
for each (var item:XML in rssXML..item)
{
  var itemTitle:String = item.title.toString();
  var itemDescription:String = item.description.toString();
  var itemLink:String = item.link.toString();
  outXML += buildItemHTML(itemTitle,
                itemDescription,
                itemLink);
}
```

前三行代码只是设置字符串变量，以表示 XML 数据的 item 属性的标题、描述和链接属性。下一行随后调用 buildItemHTML() 方法，以 XMLList 对象形式获取 HTML 数据，并以三个新的字符串变量作为参数。

## 组合 XMLList 数据

HTML 数据（XMLList 对象）具有如下形式：

```
<b>itemTitle</b>
<p>
  itemDescription
  <br />
  <a href="link">
    <font color="#008000">More...</font>
  </a>
</p>
```

方法的第一行清除默认的 XML 命名空间：

```
default xml namespace = new Namespace();
```

default xml namespace 指令具有函数块级作用域。这意味着此声明的作用域是 buildItemHTML() 方法。

下面的几行代码基于传递给该函数的字符串参数组合 XMLList：

```
var body:XMLList = new XMLList();
body += new XML("<b>" + itemTitle + "</b>");
var p:XML = new XML("<p>" + itemDescription + "</p>");

var link:XML = <a></a>;
link.@href = itemLink; // <link href="itemLinkString"></link>
link.font.@color = "#008000";
     // <font color="#008000"></font></a>
     // 0x008000 = green
link.font = "More...";

p.appendChild(<br/>);
p.appendChild(link);
body += p;
```

此 XMLList 对象表示适用于 ActionScript HTML 文本字段的字符串数据。

xmlLoaded() 方法使用 buildItemHTML() 方法的返回值并将其转换为字符串：

```
XML.prettyPrinting = false;
rssOutput = outXML.toXMLString();
```

## 提取 RSS 供给的标题并发送自定义事件

xmlLoaded() 方法基于源 RSS XML 数据中信息设置 rssTitle 字符串变量：

```
rssTitle = rssXML.channel.title.toString();
```

最后， xmlLoaded() 方法生成一个事件，通知应用程序数据已分析并且可用：

```
    dataWritten = new Event("dataWritten", true);
```

第 12 章

# 显示编程

12

ActionScript 3.0 中的显示编程用于处理出现在 Adobe Flash Player 9 的舞台上的元素。本章介绍了有关处理屏幕元素的一些基本概念。您将了解有关通过编程方式组织可视元素的详细信息。还将了解如何为显示对象创建自己的自定义类。

## 目录

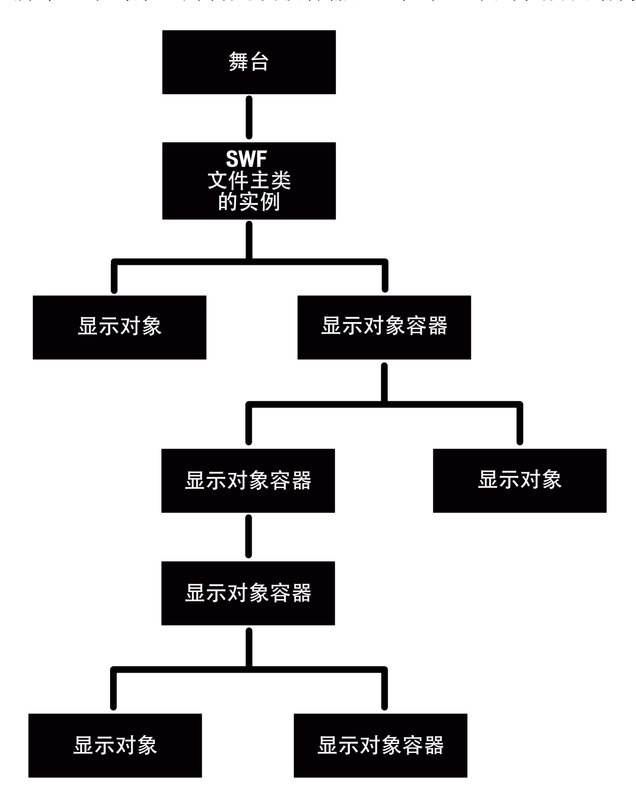
# 显示编程的基础知识

## 显示编程简介

使用 ActionScript 3.0 构建的每个应用程序都有一个由显示对象构成的层次结构，这个结构称为 “显示列表”。显示列表包含应用程序中的所有可视元素。显示元素属于下列一个或多个组：

■ 舞台

舞台是包括显示对象的基础容器。每个应用程序都有一个 Stage 对象，其中包含所有的屏幕显示对象。舞台是顶级容器，它位于显示列表层次结构的顶部：




每个 SWF 文件都有一个关联的 ActionScript 类，称为“SWF 文件的主类”。当 Flash Player 在 HTML 页中打开 SWF 文件时， Flash Player 将调用该类的构造函数，所创建的实例 （始终是一种显示对象）将添加为 Stage 对象的子级。 SWF 文件的主类始终扩展 Sprite 类 （有关详细信息，请参阅第 326 页的 “显示列表方法的优点”）。

可以通过任何 DisplayObject 实例的 `stage` 属性来访问舞台。有关详细信息，请参阅第 334 页的 “设置舞台属性”。

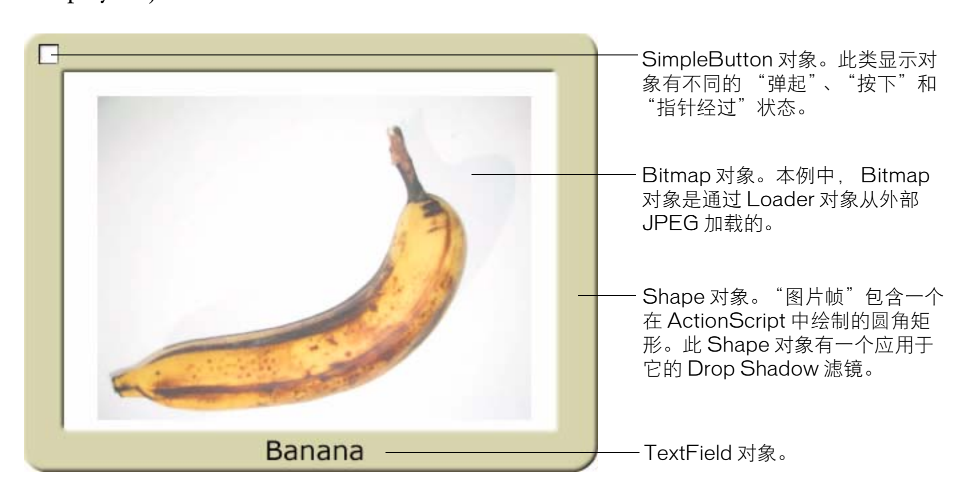
- 显示对象

  在 ActionScript 3.0 中，在应用程序屏幕上出现的所有元素都属于 “显示对象”类型。flash.display 包中包括的 DisplayObject 类是由许多其它类扩展的基类。这些不同的类表示一些不同类型的显示对象，如矢量形状、影片剪辑和文本字段等。有关这些类的概述，请参阅第 326 页的 “显示列表方法的优点”。

- 显示对象容器

  显示对象容器是一些特殊类型的显示对象，这些显示对象除了有自己的可视表示形式之外，还可以包含也是显示对象的子对象。

  DisplayObjectContainer 类是 DisplayObject 类的子类。DisplayObjectContainer 对象可以在其 “子级列表”中包含多个显示对象。例如，下图显示一种称为 Sprite 的 DisplayObjectContainer 对象，其中包含各种显示对象：




在讨论显示对象的上下文中，DisplayObjectContainer 对象又称为“显示对象容器”或简称为 “容器”。

尽管所有可视显示对象都从 DisplayObject 类继承，但每类显示对象都是 DisplayObject 类的一个特定子类。例如，有 Shape 类或 Video 类的构造函数，但没有 DisplayObject 类的构造函数。

如前所述，舞台是显示对象容器。

## 常见的显示编程任务

由于这么多的 ActionScript 编程都涉及创建和处理可视元素，因此存在许多与显示编程有关的任务。本章介绍适用于所有显示对象的常见任务，其中包括：

- 处理显示列表和显示对象容器
  - 在显示列表中添加显示对象
  - 从显示列表中删除对象
  - 在显示容器间移动对象
  - 将对象移到其它对象的前面或后面
- 处理舞台
  - 设置帧速率
  - 控制舞台缩放比例
  - 处理全屏模式
- 处理显示对象事件
- 确定显示对象的位置，包括创建拖放交互组件
- 缩放、旋转显示对象及调整其大小
- 对显示对象应用混合模式、颜色转换和透明度
- 遮罩显示对象
- 对显示对象进行动画处理
- 加载外部显示内容（如 SWF 文件或图像）

本手册中的后面几章将介绍有关处理显示对象的其它任务。这些任务包括适用于任何显示对象的任务和与特定类型的显示对象相关的任务：

- 使用 ActionScript 在显示对象上绘制矢量图形，在第 387 页的第 14 章 “使用绘图 API”中予以介绍
- 对显示对象应用几何变形，在第 371 页的第 13 章 “处理几何结构”中予以介绍
- 对显示对象应用图形滤镜效果（如模糊、发光、投影等），在第 403 页的第 15 章 “过滤显示对象”中予以介绍
- 处理特定于影片剪辑的特性，在第 425 页的第 16 章 “处理影片剪辑”中予以介绍
- 处理 TextField 对象，在第 439 页的第 17 章 “处理文本”中予以介绍
- 处理位图图形，在第 465 页的第 18 章 “处理位图”中予以介绍
- 处理视频元素，在第 477 页的第 19 章 “处理视频”中予以介绍

## 重要概念和术语

以下参考列表包含将会在本章中遇到的重要术语：

- Alpha：表示颜色中的透明度（更准确地说，是不透明度）的颜色值。例如，Alpha 通道值为 60% 的颜色只显示其最大强度的 60%，即只有 40% 是透明的。
- 位图图形 (Bitmap graphic)：在计算机中定义为彩色像素网格（行和列）的图形。通常，位图图形包括数码照片和类似图像。
- 混合模式 (Blending mode)：指定两个重叠图像的内容应如何进行交互。通常，一个图像上面的另一个不透明图像会遮盖住下面的图像，因此根本看不到该图像；但是，不同的混合模式会导致图像颜色以不同方式混合在一起，因此，生成的内容是两个图像的某种组合形式。
- 显示列表 (Display list)：由 Flash Player 呈现为可见屏幕内容的显示对象的层次结构。舞台是显示列表的根，附加到舞台或其子级之一上的所有显示对象构成了显示列表（即使并未实际呈现该对象，例如，如果它位于舞台边界以外）。
- 显示对象 (Display object)：在 Flash Player 中表示某种类型的可视内容的对象。显示列表中只能包含显示对象，所有显示对象类是 DisplayObject 类的子类。
- 显示对象容器 (Display object container)：一种特殊类型的显示对象，除了（通常）具有其自己的可视表示形式以外，它还可以包含子显示对象。
- SWF 文件的主类 (Main class of the SWF file)：为 SWF 文件中最外面的显示对象定义行为的类，从概念上讲，它是 SWF 文件本身的类。例如，在 Flash 创作环境中创建的 SWF 具有一个"主时间轴"，它包含所有其它的时间轴； SWF 文件的主类是指将主时间轴作为其实例的类。
- 蒙版 (Masking)：一种将图像的某些部分隐藏起来（或者相反，只允许显示图像的某些部分）的技术。图像的隐藏部分将变为透明，因此，将显示其下面的内容。此术语与画家的遮蔽胶带非常相似，遮蔽胶带用于防止将颜料喷到某些区域上。
- 舞台 (Stage)：一个可视容器，它是 SWF 文件中的所有可视内容的基础或背景。
- 变形 (Transformation)：对图形的可视特性进行的调整，如旋转对象、改变其缩放比例、倾斜或扭曲其形状或者改变其颜色。
- 矢量图形 (Vector graphic)：在计算机中定义为使用特定特性（如粗细、长度、大小、角度以及位置）绘制的线条和形状的图形。

## 完成本章中的示例

学习本章的过程中，您可能想要自己动手测试一些示例代码清单。由于本章是关于创建和操作可视内容，本章中的所有代码清单会实际创建可视对象并将它们显示在屏幕上；测试范例将涉及查看 Flash Player 中的结果，而不是查看前几章中的变量值。要测试本章中的代码清单，请执行以下操作：

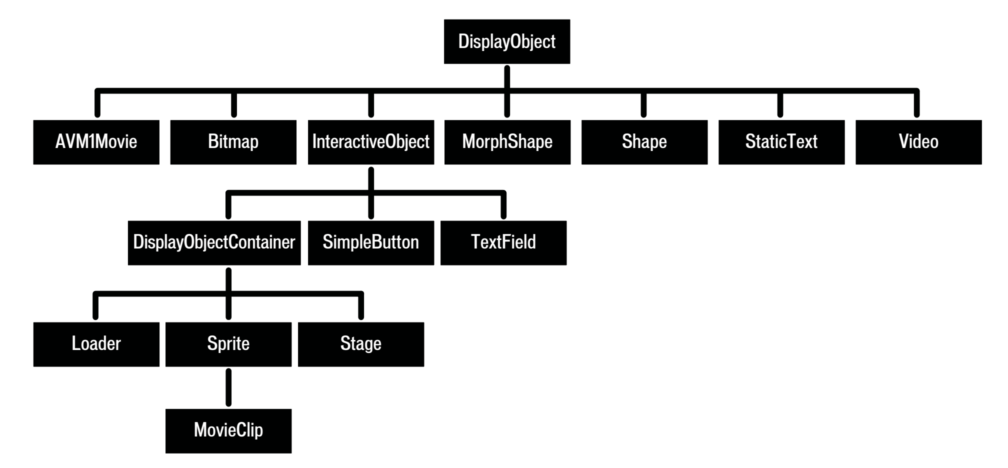
1. 创建一个空的 Flash 文档。
2. 在时间轴上选择一关键帧。
3. 打开 “动作” 面板，将代码清单复制到 “脚本” 窗格中。
4. 使用 “控制” > “测试影片” 运行程序。

您将看到显示在屏幕上的代码结果，并且所有 `trace()` 函数调用将显示在“输出”面板中。

测试示例代码清单的技术将在第 53 页的 “测试本章内的示例代码清单”中详细说明。

# 核心显示类

ActionScript 3.0 的 flash.display 包中包括可在 Flash Player 中显示的可视对象的类。下图说明了这些核心显示对象类的子类关系。




该图说明了显示对象类的类继承。请注意，其中某些类，尤其是 StaticText、 TextField 和 Video 类，不在 flash.display 包中，但它们仍然是从 DisplayObject 类继承的。

扩展 DisplayObject 类的所有类都继承该类的方法和属性。有关详细信息，请参阅第 328 页的 “DisplayObject 类的属性和方法”。

可以实例化包含在 flash.display 包中的下列类的对象：

- Bitmap — 使用 Bitmap 类可定义从外部文件加载或通过 ActionScript 呈现的位图对象。可以通过 Loader 类从外部文件加载位图。可以加载 GIF、 JPG 或 PNG 文件。还可以创建包含自定义数据的 BitmapData 对象，然后创建使用该数据的 Bitmap 对象。可以使用 BitmapData 类的方法来更改位图，无论这些位图是加载的还是在 ActionScript 中创建的。有关详细信息，请参阅第 360 页的 “加载显示对象”和第 465 页的第 18 章 “处理位图”。
- Loader — 使用 Loader 类可加载外部资源 （SWF 文件或图形）。有关详细信息，请参阅第 360 页的 “动态加载显示内容”。
- Shape — 使用 Shape 类可创建矢量图形，如矩形、直线、圆等。有关详细信息，请参阅第 387 页的第 14 章 “使用绘图 API”。
- SimpleButton — SimpleButton 对象是 Flash 按钮元件的 ActionScript 表示形式。SimpleButton 实例有 3 个按钮状态：弹起、按下和指针经过。
- Sprite — Sprite 对象可以包含它自己的图形，还可以包含子显示对象。（Sprite 类用于扩展 DisplayObjectContainer 类）。有关详细信息，请参阅第 329 页的 “处理显示对象容器”和第 387 页的第 14 章 “使用绘图 API”。
- MovieClip — MovieClip 对象是在 Flash 创作工具中创建的 ActionScript 形式的影片剪辑元件。实际上，MovieClip 与 Sprite 对象类似，不同的是它还有一个时间轴。有关详细信息，请参阅第 425 页的第 16 章 “处理影片剪辑”。

下列类不在 flash.display 包中，这些类是 DisplayObject 类的子类：

- TextField 类包括在 flash.text 包中，它是用于文本显示和输入的显示对象。有关详细信息，请参阅第 439 页的第 17 章 “处理文本”。
- Video 类包括在 flash.media 包中，它是用于显示视频文件的显示对象。有关详细信息，请参阅第 477 页的第 19 章 “处理视频”。

flash.display 包中的下列类用于扩展 DisplayObject 类，但您不能创建这些类的实例。这些类而是用作其它显示对象的父类，因此可将通用功能合并到一个类中。

- AVM1Movie — AVM1Movie 类用于表示在 ActionScript 1.0 和 2.0 中创作的已加载 SWF 文件。
- DisplayObjectContainer — Loader、Stage、Sprite 和 MovieClip 类每个都用于扩展了 DisplayObjectContainer 类。有关详细信息，请参阅第 329 页的“处理显示对象容器”。
- InteractiveObject — InteractiveObject 是用于与鼠标和键盘交互的所有对象的基类。SimpleButton、 TextField、 Video、 Loader、 Sprite、 Stage 和 MovieClip 对象是 InteractiveObject 类的所有子类。有关创建鼠标和键盘交互的详细信息，请参阅第 543 页的第 21 章 “捕获用户输入”。
- MorphShape — 这些对象是在 Flash 创作工具中创建补间形状时创建的。无法使用 ActionScript 实例化这些对象，但可以从显示列表中访问它们。

- Stage — Stage 类用于扩展 DisplayObjectContainer 类。有一个应用程序的 Stage 实例，该实例位于显示列表层次结构的顶部。要访问 Stage，请使用任何 DisplayObject 实例的 `stage` 属性。有关详细信息，请参阅第 334 页的 "设置舞台属性"。

此外，flash.text 包中的 StaticText 类也用于扩展 DisplayObject 类，但不能在代码中创建它的实例。只能在 Adobe Flash CS3 Professional 中创建静态文本字段。

# 显示列表方法的优点

在 ActionScript 3.0 中，不同类型的显示对象有不同的类。在 ActionScript 1.0 和 2.0 中，很多相同类型的对象都包括在一个类（即 MovieClip 类）中。

类的这种个性化和显示列表的层次结构有下列优点：

- 呈现方式更为有效且减少了内存使用
- 改进了深度管理
- 完整遍历显示列表
- 列表外的显示对象
- 更便于创建显示对象的子类

这些优点将在下面几部分中予以介绍。

## 呈现方式更为有效且文件较小

在 ActionScript 1.0 和 2.0 中，只可以在 MovieClip 对象中绘制形状。在 ActionScript 3.0 中，提供了可在其中绘制形状的更简单的显示对象类。由于这些 ActionScript 3.0 显示对象类并不包括 MovieClip 对象中包含的全部方法和属性，因此给内存和处理器资源造成的负担比较小。

例如，每个 MovieClip 对象都包括用于影片剪辑时间轴的属性，而 Shape 对象则不包括。用于管理时间轴的属性会使用大量的内存和处理器资源。在 ActionScript 3.0 中，使用 Shape 对象可提高性能。与更复杂的 MovieClip 对象相比，Shape 对象的开销更少。Flash Player 并不需要管理未使用的 MovieClip 属性，因此提高了速度，还减少了对象使用的内存空间。

## 改进了深度管理

在 ActionScript 1.0 和 2.0 中，通过线性深度管理方案和方法（如 `getNextHighestDepth()`）进行深度管理。

在 ActionScript 3.0 中提供了 DisplayObjectContainer 类，该类提供用于管理显示对象深度的更便捷的方法和属性。

在 ActionScript 3.0 中，当您将显示对象移到 DisplayObjectContainer 实例的子级列表中的新位置时，显示对象容器中的其它子级会自动重新定位并在显示对象容器中分配相应的子索引位置。

此外，在 ActionScript 3.0 中，总是可以发现任何显示对象容器中的所有子对象。每个 DisplayObjectContainer 实例都有 `numChildren` 属性，用于列出显示对象容器中的子级数。由于显示对象容器的子级列表始终是索引列表，因此可以检查列表中从索引位置 0 到最后一个索引位置 (`numChildren - 1`) 的所有对象。这不适用于 ActionScript 1.0 和 2.0 中 MovieClip 对象的方法和属性。

在 ActionScript 3.0 中，可以按顺序轻松遍历显示列表；显示对象容器子级列表的索引编号没有中断。遍历显示列表和管理对象深度比在 ActionScript 1.0 和 2.0 中更容易。在 ActionScript 1.0 和 2.0 中，影片剪辑可以包含深度顺序有间歇中断的对象，这可能导致难以遍历对象列表。在 ActionScript 3.0 中，显示对象容器的每个子级列表都在内部缓存为一个数组，这样按索引查找的速度就非常快。遍历显示对象容器所有子级的速度也非常快。

在 ActionScript 3.0 中，还可以通过使用 DisplayObjectContainer 类的 `getChildByName()` 方法来访问显示对象容器中的子级。

## 完整遍历显示列表

在 ActionScript 1.0 和 2.0 中，无法访问在 Flash 创作工具中绘制的某些对象（如矢量形状）。在 ActionScript 3.0 中，可以访问显示列表中的所有对象，包括使用 ActionScript 创建的对象以及在 Flash 创作工具中创建的所有显示对象。有关详细信息，请参阅第 333 页的 “遍历显示列表”。

## 列表外的显示对象

在 ActionScript 3.0 中，可以创建不在可视显示列表中的显示对象。这些对象称为 “列表外的”显示对象。仅当调用已添加到显示列表中的 DisplayObjectContainer 实例的 `addChild()` 或 `addChildAt()` 方法时，才会将显示对象添加到可视显示列表中。

可以使用列表外的显示对象来组合复杂的显示对象，如有多个显示对象容器（包含多个显示对象）的那些对象。通过将显示对象放在列表外，可以组合复杂的对象，而不需要占用处理时间来呈现这些显示对象。然后在需要时可以在显示列表中添加列表外的显示对象。此外，可以随意将显示对象容器的子级移入和移出显示列表以及移到显示列表中的任何需要位置。

## 更便于创建显示对象的子类

在 ActionScript 1.0 和 2.0 中，通常必须在 SWF 文件中添加新的 MovieClip 对象才能创建基本形状或显示位图。在 ActionScript 3.0 中，DisplayObject 类包括许多内置子类，包括 Shape 和 Bitmap。由于 ActionScript 3.0 中的类更专用于特定类型的对象，因此更易于创建内置类的基本子类。

例如，要在 ActionScript 2.0 中绘制一个圆，可以在实例化自定义类的对象时创建用于扩展 MovieClip 类的 CustomCircle 类。但是，该类还另外包括 MovieClip 类中不应用于该类的许多属性和方法（如 `totalFrames`）。但在 ActionScript 3.0 中，可以创建用于扩展 Shape 对象的 CustomCircle 类，但该类不包括 MovieClip 类中包含的不相关的属性和方法。下面的代码显示了 CustomCircle 类的一个示例：

```
import flash.display.*;

private class CustomCircle extends Shape
{
  var xPos:Number;
  var yPos:Number;
  var radius:Number;
  var color:uint;
  public function CustomCircle(xInput:Number,
                       yInput:Number,
                       rInput:Number,
                       colorInput:uint)
  {
    xPos = xInput;
    yPos = yInput;
    radius = rInput;
    color = colorInput;
    this.graphics.beginFill(color);
    this.graphics.drawCircle(xPos, yPos, radius);
  }
}
```

# 处理显示对象

现在您已了解了舞台、显示对象、显示对象容器和显示列表的基本概念，本部分将为您提供有关在 ActionScript 3.0 中处理显示对象的一些更具体的信息。

## DisplayObject 类的属性和方法

所有显示对象都是 DisplayObject 类的子类，同样它们还会继承 DisplayObject 类的属性和方法。继承的属性是适用于所有显示对象的基本属性。例如，每个显示对象都有 `x` 属性和 `y` 属性，用于指定对象在显示对象容器中的位置。

您不能使用 DisplayObject 类构造函数来创建 DisplayObject 实例。必须创建另一种对象（属于 DisplayObject 类的子类的对象，如 Sprite）才能使用 new 运算符来实例化对象。此外，如果要创建自定义显示对象类，还必须创建具有可用构造函数的其中一个显示对象子类的子类（如 Shape 类或 Sprite 类）。有关详细信息，请参阅《ActionScript 3.0 语言和组件参考》中的 DisplayObject 类说明。

## 在显示列表中添加显示对象

实例化显示对象时，在将显示对象实例添加到显示列表上的显示对象容器之前，显示对象不会出现屏幕上（即在舞台上）。例如，在下面的代码中，如果省略了最后一行代码，则 myText TextField 对象不可见。在最后一行代码中，this 关键字必须引用已添加到显示列表中的显示对象容器。

```
import flash.display.*;
import flash.text.TextField;
var myText:TextField = new TextField();
myText.text = "Buenos dias.";
this.addChild(myText);
```

当在舞台上添加任何可视元素时，该元素会成为 Stage 对象的 “子级”。应用程序中加载的第一个 SWF 文件（例如，HTML 页中嵌入的文件）会自动添加为 Stage 的子级。它可以是扩展 Sprite 类的任何类型的对象。

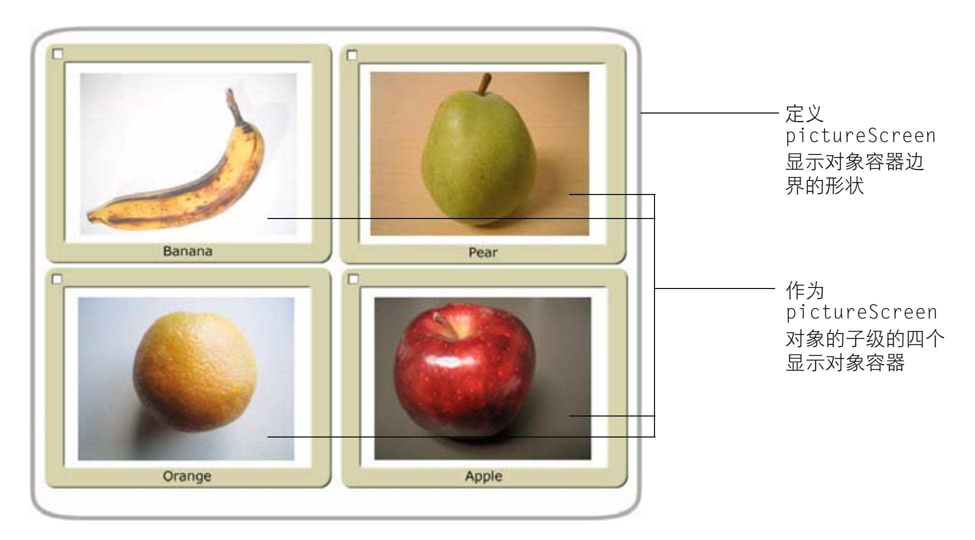
“不是”使用 ActionScript 创建的任何显示对象（例如，通过在 Adobe Flex Builder 2 中添加 MXML 标签或在 Flash 的舞台上放置某项而创建任何对象），都会添加到显示列表中。尽管没有通过 ActionScript 添加这些显示对象，但仍可通过 ActionScript 访问它们。例如，下面的代码将调整在创作工具中（不是通过 ActionScript）添加的名为 button1 的对象的宽度：

```
button1.width = 200;
```

## 处理显示对象容器

如果从显示列表中删除某个 DisplayObjectContainer 对象，或者以其它某种方式移动该对象或对其进行变形处理，则会同时删除、移动 DisplayObjectContainer 中的每个显示对象或对其进行变形处理。

显示对象容器本身就是一种显示对象，它可以添加到其它显示对象容器中。例如，下图显示的是显示对象容器 pictureScreen，它包含一个轮廓形状和四个其它显示对象容器（类型为 PictureFrame）：




要使某一显示对象出现在显示列表中，必须将该显示对象添加到显示列表上的显示对象容器中。使用容器对象的 addChild() 方法或 addChildAt() 方法可执行此操作。例如，如果下面的代码没有最后一行，将不会显示 myTextField 对象：

```
var myTextField:TextField = new TextField();
myTextField.text = "hello";
this.root.addChild(myTextField);
```

在此代码范例中，this.root 指向包含该代码的 MovieClip 显示对象容器。在实际代码中，可以指定其它容器。

使用 addChildAt() 方法可将子级添加到显示对象容器的子级列表中的特定位置。子级列表中这些从 0 开始的索引位置与显示对象的分层（从前到后顺序）有关。例如，请考虑下列三个显示对象。每个对象都是从称为 Ball 的自定义类创建的。



使用 addChildAt() 方法可以调整这些显示对象在容器中的分层。例如，请考虑使用以下代码：

```
ball_A = new Ball(0xFFCC00, "a");
ball_A.name = "ball_A";
ball_A.x = 20;
ball_A.y = 20;
container.addChild(ball_A);

ball_B = new Ball(0xFFCC00, "b");
ball_B.name = "ball_B";
ball_B.x = 70;
ball_B.y = 20;
container.addChild(ball_B);

ball_C = new Ball(0xFFCC00, "c");
ball_C.name = "ball_C";
ball_C.x = 40;
ball_C.y = 60;
container.addChildAt(ball_C, 1);
```

执行此代码后，显示对象在 container DisplayObjectContainer 对象中的定位如下所示。请注意对象的分层。




要重新将对象定位到显示列表的顶部，只需重新将其添加到列表中。例如，在前面的代码后，要将 ball_A 移到堆栈的顶部，请使用下面的代码行：

```
container.addChild(ball_A);
```

此代码可有效地将 ball_A 从它在 container 的显示列表中的位置删除，然后将它重新添加到列表的顶部，最终的结果是将它移到堆栈的顶部。

可以使用 getChildAt() 方法来验证显示对象的图层顺序。getChildAt() 方法根据您向容器传递的索引编号返回容器的子对象。例如，下面的代码显示 container DisplayObjectContainer 对象的子级列表中不同位置的显示对象的名称：

```
trace(container.getChildAt(0).name); // ball_A
trace(container.getChildAt(1).name); // ball_C
trace(container.getChildAt(2).name); // ball_B
```

如果从父容器的子级列表中删除了一个显示对象，则列表中位置较高的每一个元素在子索引中会分别下移一个位置。例如，接着前面的代码，下面的代码显示如果删除子级列表中位置较低的一个显示对象，container DisplayObjectContainer 中位置 2 的显示对象如何移到位置 1：

```
container.removeChild(ball_C);
trace(container.getChildAt(0).name); // ball_A
trace(container.getChildAt(1).name); // ball_B
```

removeChild() 和 removeChildAt() 方法并不完全删除显示对象实例。这两种方法只是从容器的子级列表中删除显示对象实例。该实例仍可由另一个变量引用。（请使用 delete 运算符完全删除对象。）

由于显示对象只有一个父容器，因此只能在一个显示对象容器中添加显示对象的实例。例如，下面的代码说明了显示对象 tf1 只能存在于一个容器中（本例中为 Sprite，它扩展 DisplayObjectContainer 类）：

```
tf1:TextField = new TextField();
tf2:TextField = new TextField();
tf1.name = "text 1";
tf2.name = "text 2";

container1:Sprite = new Sprite();
container2:Sprite = new Sprite();

container1.addChild(tf1);
container1.addChild(tf2);
container2.addChild(tf1);

trace(container1.numChildren); // 1
trace(container1.getChildAt(0).name); // 文本 2
trace(container2.numChildren); // 1
trace(container2.getChildAt(0).name); // 文本 1
```

如果将在第一个显示对象容器中包含的某一显示对象添加到另一个显示对象容器中，则会从第一个显示对象容器的子级列表中删除该显示对象。

除了上面介绍的方法之外，DisplayObjectContainer 类还定义了用于处理子显示对象的几个方法，其中包括：

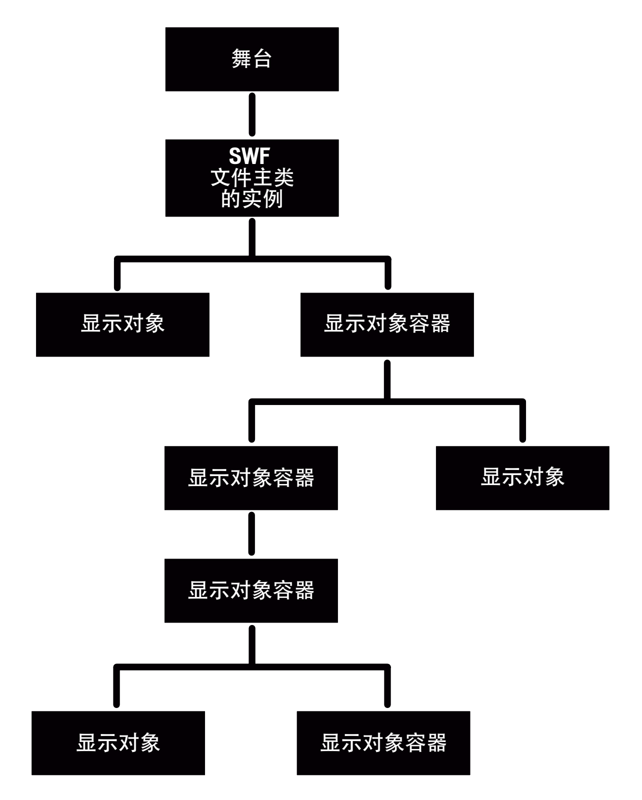
- contains()：确定显示对象是否是 DisplayObjectContainer 的子级。
- getChildByName()：按名称检索显示对象。
- getChildIndex()：返回显示对象的索引位置。
- setChildIndex()：更改子显示对象的位置。
- swapChildren()：交换两个显示对象的前后顺序。
- swapChildrenAt()：交换两个显示对象的前后顺序（由其索引值指定）。

有关详细信息，请参阅《ActionScript 3.0 语言和组件参考》中的相关条目。

注意，不在显示列表中的显示对象（也就是不包括在舞台子级的显示对象容器中的显示对象）被称为“列表外”的显示对象。

## 遍历显示列表

正如所看到的，显示列表是一个树结构。树的顶部是舞台，它可以包含多个显示对象。那些本身就是显示对象容器的显示对象可以包含其它显示对象或显示对象容器。




DisplayObjectContainer 类包括通过显示对象容器的子级列表遍历显示列表的属性和方法。例如，考虑下面的代码，其中在 `container` 对象（该对象为 Sprite，Sprite 类用于扩展 DisplayObjectContainer 类）中添加了两个显示对象 `title` 和 `pict`：

```
var container:Sprite = new Sprite();
var title:TextField = new TextField();
title.text = "Hello";
var pict:Loader = new Loader();
var url:URLRequest = new URLRequest("banana.jpg");
pict.load(url);
pict.name = "banana loader";
```

```
container.addChild(title);
container.addChild(pict);
```

getChildAt() 方法返回显示列表中特定索引位置的子级：

```
trace(container.getChildAt(0) is TextField); // true
```

您也可以按名称访问子对象。每个显示对象都有一个名称属性，如果没有指定该属性，Flash Player 会指定一个默认值，如 "instance1"。例如，下面的代码说明了如何使用 getChildByName() 方法来访问名为 "banana loader" 的子显示对象：

```
trace(container.getChildByName("banana loader") is Loader); // true
```

与使用 getChildAt() 方法相比，使用 getChildByName() 方法会导致性能降低。

由于显示对象容器可以包含其它显示对象容器作为其显示列表中的子对象，因此您可将应用程序的完整显示列表作为树来遍历。例如，在前面说明的代码摘录中，完成 pict Loader 对象的加载操作后，pict 对象将加载一个子显示对象，即位图。要访问此位图显示对象，可以编写 pict.getChildAt(0)。还可以编写 container.getChildAt(0).getChildAt(0)（由于 container.getChildAt(0) == pict）。

下面的函数提供了显示对象容器中显示列表的缩进式 trace() 输出：

```
function traceDisplayList(container:DisplayObjectContainer,
                          indentString:String = ""):void
{
  var child:DisplayObject;
  for (var i:uint=0; i < container.numChildren; i++)
  {
    child = container.getChildAt(i);
    trace(indentString, child, child.name);
    if (container.getChildAt(i) is DisplayObjectContainer)
    {
      traceDisplayList(DisplayObjectContainer(child), indentString + "")
    }
  }
}
```

## 设置舞台属性

Stage 类用于覆盖 DisplayObject 类的大多数属性和方法。如果调用其中一个已覆盖的属性或方法，Flash Player 会引发异常。例如，Stage 对象不具有 x 或 y 属性，因为作为应用程序的主容器，该对象的位置是固定的。x 和 y 属性是指显示对象相对于其容器的位置，由于舞台没有包含在其它显示对象容器中，因此这些属性不适用。

| 注意 | 如果显示对象与加载的第一个 SWF 文件不在同一个安全沙箱中，则 Stage 类的某些属性和方法不适用于这些显示对象。有关详细信息，请参阅第 670 页的 “Stage 安全性”。 |
|---|---|

## 控制回放帧速率

Stage 类的 `framerate` 属性用于设置加载到应用程序中的所有 SWF 文件的帧速率。有关详细信息，请参阅《ActionScript 3.0 语言和组件参考》。

## 控制舞台缩放比例

当调整 Flash Player 屏幕的大小时， Flash Player 会自动调整舞台内容来加以补偿。Stage 类的 `scaleMode` 属性可确定如何调整舞台内容。此属性可以设置为四个不同值，如 flash.display.StageScaleMode 类中的常量所定义。

对于 `scaleMode` 的三个值（`StageScaleMode.EXACT_FIT`、`StageScaleMode.SHOW_ALL` 和 `StageScaleMode.NO_BORDER`）， Flash Player 将缩放舞台的内容以容纳在舞台边界内。三个选项在确定如何完成缩放时是不相同的。

- `StageScaleMode.EXACT_FIT` 按比例缩放 SWF。
- `StageScaleMode.SHOW_ALL` 确定是否显示边框（就像在标准电视上观看宽屏电影时显示的黑条）。
- `StageScaleMode.NO_BORDER` 确定是否可以部分裁切内容。

或者，如果将 `scaleMode` 设置为 `StageScaleMode.NO_SCALE`，则当查看者调整 Flash Player 窗口大小时，舞台内容将保持定义的大小。仅在缩放模式中， Stage 类的 `width` 和 `height` 属性才可用于确定 Flash Player 窗口调整大小后的实际像素尺寸。（在其它缩放模式中，`stageWidth` 和 `stageHeight` 属性始终反映的是 SWF 的原始宽度和高度。）此外，当 `scaleMode` 设置为 `StageScaleMode.NO_SCALE` 并且调整了 SWF 文件大小时，将调度 Stage 类的 `resize` 事件，允许您进行相应地调整。

因此，将 `scaleMode` 设置为 `StageScaleMode.NO_SCALE` 可以更好地控制如何根据需要调整屏幕内容以适合窗口大小。例如，在包含视频和控制栏的 SWF 中，您可能希望在调整舞台大小时控制栏的大小保持不变，而仅更改视频窗口大小以适应舞台大小的更改。以下示例中演示了这一点：

```
// videoScreen 是一个包含视频的显示对象（例如，视频实例）
//；它位于舞台左上角，并且
// 在调整 SWF 大小时其大小也应调整。

// controlBar 是一个包含多个按钮的显示对象（例如， Sprite），
// 它应位于舞台（在 videoScreen 的下方）的左下角，
// 在调整 SWF 大小时其大小将不会
// 调整。

import flash.display.Stage;
import flash.display.StageAlign;
import flash.display.StageScaleMode;
import flash.events.Event;

var swfStage:Stage = videoScreen.stage;
```

```
swfStage.scaleMode = StageScaleMode.NO_SCALE;
swfStage.align = StageAlign.TOP_LEFT;

function resizeDisplay(event:Event):void
{
    var swfWidth:int = swfStage.stageWidth;
    var swfHeight:int = swfStage.stageHeight;

    // Resize the video window.
    var newVideoHeight:Number = swfHeight - controlBar.height;
    videoScreen.height = newVideoHeight;
    videoScreen.scaleX = videoScreen.scaleY;

    // Reposition the control bar.
    controlBar.y = newVideoHeight;
}

swfStage.addEventListener(Event.RESIZE, resizeDisplay);
```

## 处理全屏模式

使用全屏模式可令 SWF 填充查看器的整个显示器，没有任何边框、菜单栏等。Stage 类的 `displayState` 属性用于切换 SWF 的全屏模式。可以将 `displayState` 属性设置为由 flash.display.StageDisplayState 类中的常量定义的其中一个值。要打开全屏模式，请将 `displayState` 设置为 `StageDisplayState.FULL_SCREEN`：

```
// mySprite 是一个 Sprite 实例，已添加到显示列表中
mySprite.stage.displayState = StageDisplayState.FULL_SCREEN;
```

要退出全屏模式，请将 `displayState` 属性设置为 `StageDisplayState.NORMAL`：

```
mySprite.stage.displayState = StageDisplayState.NORMAL;
```

此外，用户可以通过将焦点切换到其它窗口或使用以下组合键之一来选择退出全屏模式：Esc（所有平台）、Ctrl-W (Windows)、Command-W (Mac) 或 Alt-F4 (Windows)。

全屏模式的舞台缩放行为与正常模式下的相同；缩放比例由 Stage 类的 `scaleMode` 属性控制。通常，如果将 `scaleMode` 属性设置为 `StageScaleMode.NO_SCALE`，则 Stage 的 `stageWidth` 和 `stageHeight` 属性将发生更改，以反映由 SWF 占用的屏幕区域的大小（在本例中为整个屏幕）。

打开或关闭全屏模式时，可以使用 Stage 类的 `fullScreen` 事件来检测和响应。例如，进入或退出全屏模式时，您可能需要重新定位、添加或删除屏幕中的项目，如本例中所示：

```
import flash.events.FullScreenEvent;

function fullScreenRedraw(event:FullScreenEvent):void
{
    if (event.fullScreen)
    {
        // 删除输入文本字段。
```

```
        // 添加关闭全屏模式的按钮。
    }
    else
    {
        // 重新添加输入文本字段。
        // 删除关闭全屏模式的按钮。
    }
}

mySprite.stage.addEventListener(FullScreenEvent.FULL_SCREEN,
    fullScreenRedraw);
```

正如此代码所示，`fullScreen` 事件的事件对象是 flash.events.FullScreenEvent 类的实例，它包含指示是启用 (`true`) 还是禁用 (`false`) 全屏模式的 `fullScreen` 属性。

在 ActionScript 中处理全屏模式时，需要记住以下注意事项：

- 只能通过 ActionScript 响应鼠标单击（包括右键单击）或按键才能启动全屏模式。
- 对于有多个显示器的用户， SWF 内容将展开且只填充一个显示器。 Flash Player 使用度量信息来确定哪个显示器包含 SWF 的最大部分内容，然后使用该显示器提供全屏模式。
- 对于 HTML 页中嵌入的 SWF 文件，嵌入 Flash Player 的 HTML 代码必须包括名为 `allowFullScreen` 且值为 `true` 的 `param` 标签和 `embed` 属性，如下所示：

  ```
  <object>
      ...
      <param name="allowFullScreen" value="true" />
      <embed ... allowfullscreen="true" />
  </object>
  ```

  如果要在网页中使用 JavaScript 来生成 SWF 嵌入标签，则必须更改 JavaScript 以添加 `allowFullScreen param` 标签和属性。例如，如果 HTML 页使用 `AC_FL_RunContent()` 函数（由 Flex Builder 和 Flash 生成的 HTML 页使用），则应在该函数调用中添加 `allowFullScreen` 参数，如下所示：

  ```
  AC_FL_RunContent(
      ...
      'allowFullScreen','true',
      ...
      ); //end AC code
  ```

  这不适用于在独立 Flash Player 中运行的 SWF 文件。
- 在全屏模式下将禁用所有与键盘有关的 ActionScript，如 TextField 实例中的键盘事件和文本输入。用于关闭全屏模式的键盘快捷键除外。

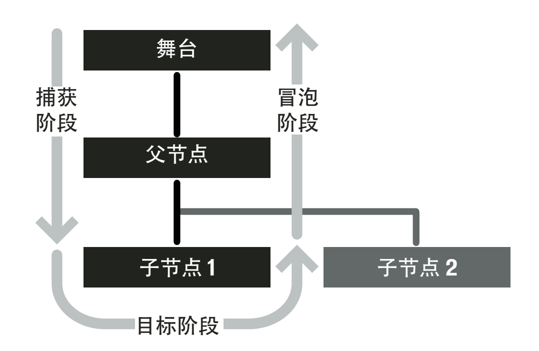
还有其它几个需要了解的与安全有关的限制。这些将在第 660 页的 “安全沙箱” 中予以介绍。

## 处理显示对象的事件

DisplayObject 类从 EventDispatcher 类继承。这意味着，每个显示对象都可完全参与到事件模型中（在第 267 页的第 10 章 “处理事件”中予以介绍）。每个显示对象可使用其 `addEventListener()` 方法（继承自 EventDispatcher 类）来侦听特定的事件，但只有侦听对象是该事件的事件流的一部分时才能实现。

当 Flash Player 调度某个事件对象时，该事件对象会执行从舞台到发生事件的显示对象的往返行程。例如，如果用户单击名为子级 1 的显示对象， Flash Player 会从舞台通过显示列表层次结构将事件对象向下调度到子级 1 显示对象。

从概念上说，事件流分为三个阶段，如下图所示：




有关详细信息，请参阅第 267 页的第 10 章 “处理事件”。

处理显示对象事件时需要记住的一个重要问题是：从显示列表中删除显示对象时，事件侦听器的存在将会对是否从内存中自动删除显示对象（垃圾回收）产生影响。如果显示对象拥有订阅为其事件的侦听器的对象，即使从显示列表中删除了显示对象，也不会从内存中删除显示对象，因为显示对象仍然拥有到这些侦听器对象的引用。有关详细信息，请参阅第 283 页的 “管理事件侦听器”。

## 选择 DisplayObject 子类

如果有多个可供选择的选项，处理显示对象时要作出的一个重要决策是：每个显示对象的用途是什么。以下原则可以帮助您作出决策。无论是需要类实例，还是选择要创建的类的基类，都可以应用这些建议：

- 如果不需要可作为其它显示对象的容器的对象（即只需要用作独立屏幕元素的对象），请根据使用目的选择 DisplayObject 或 InteractiveObject 两个子类中的一个：
  - 用于显示位图图像的 Bitmap。
  - 用于添加文本的 TextField。
  - 用于显示视频的 Video。

- 用于绘制屏幕内容的 “画布”的 Shape。特别是，如果要创建用于在屏幕上绘制形状的实例，而且该实例不是其它显示对象的容器，则使用 Shape 比使用 Sprite 或 MovieClip 有明显的性能优势。
- 用于 Flash 具体创作项的 MorphShape、StaticText 或 SimpleButton。（无法以编程方式创建这些类的实例，但可以通过创建这些数据类型的变量来引用使用 Flash 创作程序创建的项目。）

■ 如果需要使用变量来引用主舞台，请使用 Stage 类作为其数据类型。

■ 如果需要容器来加载外部 SWF 文件或图像文件，请使用 Loader 实例。加载的内容将作为 Loader 实例的子级添加到显示列表中。其数据类型将取决于加载内容的性质，如下所示：
  - 加载的图像将是 Bitmap 实例。
  - 使用 ActionScript 3.0 编写的已加载 SWF 文件将是 Sprite 或 MovieClip 实例（或这些类的子类的实例，由内容创建者指定）。
  - 使用 ActionScript 1.0 或 ActionScript 2.0 编写的已加载 SWF 文件将是 AVM1Movie 实例。

■ 如果需要将一个对象用作其它显示对象的容器（无论是否还要使用 ActionScript 在显示对象上进行绘制），请选择其中一个 DisplayObjectContainer 子类：
  - 如果对象是只使用 ActionScript 创建的，或者如果对象作为只使用 ActionScript 创建和处理的自定义显示对象的基类，请选择 Sprite。
  - 如果要通过创建变量来引用在 Flash 创作工具中创建的影片剪辑元件，请选择 MovieClip。

■ 如果要创建的类与 Flash 库中的影片剪辑元件关联，请选择其中一个 DisplayObjectContainer 子类作为该类的基类：
  - 如果关联的影片剪辑元件在多个帧上有内容，请选择 MovieClip
  - 如果关联的影片剪辑元件仅在第一帧上有内容，请选择 Sprite

# 处理显示对象

无论选择使用哪个显示对象，都会有许多的操作，这些操作作为屏幕上显示的一些元素是所有显示对象共有的。例如，可以在屏幕上确定所有显示对象的位置、前后移动显示对象的堆叠顺序、缩放或旋转显示对象等。由于所有显示对象都从它们共有的基类 (DisplayObject) 继承了此功能，因此无论是要处理 TextField 实例、Video 实例、Shape 实例还是其它任何显示对象，此功能的行为都相同。以下部分将详细说明几个常用显示对象操作。

## 改变位置

对任何显示对象进行的最基本操作是确定显示对象在屏幕上的位置。要设置显示对象的位置，请更改对象的 x 和 y 属性。

```
myShape.x = 17;
myShape.y = 212;
```

显示对象定位系统将舞台视为一个笛卡尔坐标系（带有水平 x 轴和垂直 y 轴的常见网格系统）。坐标系的原点（x 和 y 轴相交的 0,0 坐标）位于舞台的左上角。从原点开始， x 轴的值向右为正，向左为负，而 y 轴的值向下为正，向上为负（与典型的图形系统相反）。例如，通过前面的代码行可以将对象 myShape 移到 x 轴坐标 17（原点向右 17 个像素）和 y 轴坐标 212（原点向下 212 个像素）。

默认情况下，当使用 ActionScript 创建显示对象时，x 和 y 属性均设置为 0，从而可将对象放在其父内容的左上角。

### 改变相对于舞台的位置

一定要记住 x 和 y 属性始终是指显示对象相对于其父显示对象坐标轴的 0,0 坐标的位置，这一点很重要。因此，对于包含在 Sprite 实例内的 Shape 实例（如圆），如果将 Shape 对象的 x 和 y 属性设置为 0，则会将圆放在 Sprite 的左上角，该位置不一定是舞台的左上角。要确定对象相对于全局舞台坐标的位置，可以使用任何显示对象的 globalToLocal() 方法将坐标从局部（舞台）坐标转换为本地（显示对象容器）坐标，如下所示：

```
// 将形状定位到舞台左上角，
// 无论其父级位于什么位置。

// 创建 Sprite，确定的位置为 x: 200 和 y: 200。
var mySprite:Sprite = new Sprite();
mySprite.x = 200;
mySprite.y = 200;
this.addChild(mySprite);

// 在 Sprite 的 0,0 坐标处绘制一个点作为参考。
mySprite.graphics.lineStyle(1, 0x000000);
mySprite.graphics.beginFill(0x000000);
mySprite.graphics.moveTo(0, 0);
mySprite.graphics.lineTo(1, 0);
mySprite.graphics.lineTo(1, 1);
mySprite.graphics.lineTo(0, 1);
mySprite.graphics.endFill();

// 创建圆 Shape 实例。
var circle:Shape = new Shape();
mySprite.addChild(circle);

// 在 Shape 中绘制半径为 50 且中心点的 x 和 y 坐标均为 50 的圆。
```

```
circle.graphics.lineStyle(1, 0x000000);
circle.graphics.beginFill(0xff0000);
circle.graphics.drawCircle(50, 50, 50);
circle.graphics.endFill();

// 移动 Shape，使其左上角位于舞台的 0，0 坐标处。
var stagePoint:Point = new Point(0, 0);
var targetPoint:Point = mySprite.globalToLocal(stagePoint);
circle.x = targetPoint.x;
circle.y = targetPoint.y;
```

同样，可以使用 DisplayObject 类的 `localToGlobal()` 方法将本地坐标转换为舞台坐标。

## 创建拖放交互组件

移动显示对象的一个常见理由是要创建拖放交互组件，这样当用户单击某个对象时，在松开鼠标按键之前，该对象会随着鼠标的移动而移动。在 ActionScript 中可以采用两种方法创建拖放交互组件。在每种情况下，都会使用两个鼠标事件：按下鼠标按键时，通知对象跟随鼠标光标；松开鼠标按键时，通知对象停止跟随鼠标光标。

第一方法使用 `startDrag()` 方法，它比较简单，但限制较多。按下鼠标按键时，将调用要拖动的显示对象的 `startDrag()` 方法。松开鼠标按键时，将调用 `stopDrag()` 方法。

```
// 此代码使用 startDrag()
// 技术创建拖放交互组件。
// 正方形是一个 DisplayObject（例如 MovieClip 或 Sprite 实例）。

import flash.events.MouseEvent;

// 按下鼠标按键时会调用此函数。
function startDragging(event:MouseEvent):void
{
  square.startDrag();
}

// 松开鼠标按键时会调用此函数。
function stopDragging(event:MouseEvent):void
{
  square.stopDrag();
}

square.addEventListener(MouseEvent.MOUSE_DOWN, startDragging);
square.addEventListener(MouseEvent.MOUSE_UP, stopDragging);
```

这种方法有一个非常大的限制：每次只能使用 startDrag() 拖动一个项目。如果正在拖动一个显示对象，然后对另一个显示对象调用了 startDrag() 方法，则第一个显示对象会立即停止跟随鼠标。例如，如果 startDragging() 函数如下发生了更改，则只拖动 circle 对象，而不管 square.startDrag() 方法调用：

```
function startDragging(event:MouseEvent):void
{
  square.startDrag();
  circle.startDrag();
}
```

由于每次只能使用 startDrag() 拖动一个对象，因此，可以对任何显示对象调用 stopDrag() 方法，这会停止当前正在拖动的任何对象。

如果需要拖动多个显示对象，或者为了避免多个对象可能会使用 startDrag() 而发生冲突，最好使用鼠标跟随方法来创建拖动效果。通过这种技术，当按下鼠标按键时，会将函数作为舞台的 mouseMove 事件的侦听器来订阅。然后，每次鼠标移动时都会调用此函数，它将使所拖动的对象跳到鼠标所在的 x,y 坐标。松开鼠标按键后，取消此函数作为侦听器的订阅，这意味着鼠标移动时不再调用该函数且对象停止跟随鼠标光标。下面是演示说明此技术的一些代码：

```
// 此代码使用鼠标跟随
// 技术创建拖放交互组件。
// 圆是一个 DisplayObject（例如 MovieClip 或 Sprite 实例）。

import flash.events.MouseEvent;

var offsetX:Number;
var offsetY:Number;

// 按下鼠标按键时会调用此函数。
function startDragging(event:MouseEvent):void
{
  // 记录按下鼠标按键时光标的位置
  // 与按下鼠标按键时圆的 x，y
  // 坐标之间的差异（偏移量）。
  offsetX = event.stageX - circle.x;
  offsetY = event.stageY - circle.y;

  // 通知 Flash Player 开始侦听 mouseMove 事件
  stage.addEventListener(MouseEvent.MOUSE_MOVE, dragCircle);
}

// 松开鼠标按键时会调用此函数。
function stopDragging(event:MouseEvent):void
{
  // 通知 Flash Player 停止侦听 mouseMove 事件。
  stage.removeEventListener(MouseEvent.MOUSE_MOVE, dragCircle);
}
```

```
// 只要按下鼠标按键，
// 每次移动鼠标时都会调用此函数。
function dragCircle(event:MouseEvent):void
{
   // 将圆移到光标的位置，从而保持
   // 光标的位置和拖动对象的位置
   // 之间的偏移量。
   circle.x = event.stageX - offsetX;
   circle.y = event.stageY - offsetY;

   // 指示 Flash Player 在此事件后刷新屏幕。
   event.updateAfterEvent();
}

circle.addEventListener(MouseEvent.MOUSE_DOWN, startDragging);
circle.addEventListener(MouseEvent.MOUSE_UP, stopDragging);
```

除了使显示对象跟随鼠标光标之外，拖放交互组件的共有部分还包括将拖动对象移到显示对象的前面，以使拖动对象好像浮动在所有其它对象之上。例如，假定您有两个对象（圆和正方形），它们都有一个拖放交互组件。如果圆在显示列表中出现在正方形之下，您单击并拖动圆时光标会出现在正方形之上，圆好像在正方形之后滑动，这样中断了拖放视觉效果。您可以使用拖放交互组件避免这一点，以便在单击圆时圆移到显示列表的顶部，使圆会始终出现在其它任何内容的顶部。

下面的代码（摘自前面的示例）用于创建两个显示对象（圆和正方形）的拖放交互组件。只要在任一个显示对象上按下鼠标按键，该显示对象就会移到舞台显示列表的顶部，所以拖动的项目始终出现在顶部。新代码或以前代码中经过更改的代码用粗体显示。

```
// 此代码使用鼠标跟随
// 技术创建拖放交互组件。
// 圆和正方形是 DisplayObject（例如 MovieClip 或 Sprite
// 实例）。

import flash.display.DisplayObject;
import flash.events.MouseEvent;

var offsetX:Number;
var offsetY:Number;
var draggedObject:DisplayObject;

// 按下鼠标按键时会调用此函数。
function startDragging(event:MouseEvent):void
{
   // 记住正在拖动的对象
   draggedObject = DisplayObject(event.target);

   // 记录按下鼠标按键时光标的位置
   // 与按下鼠标按键时拖动的对象的 x，y 坐标
   // 之间的差异（偏移量）。
```

```
    offsetX = event.stageX - draggedObject.x;
    offsetY = event.stageY - draggedObject.y;

    // 将所选对象移到显示列表的顶部
    stage.addChild(draggedObject);

    // 通知 Flash Player 开始侦听 mouseMove 事件。
    stage.addEventListener(MouseEvent.MOUSE_MOVE, dragObject);
}

// 松开鼠标按键时会调用此函数。
function stopDragging(event:MouseEvent):void
{
    // 通知 Flash Player 停止侦听 mouseMove 事件。
    stage.removeEventListener(MouseEvent.MOUSE_MOVE, dragObject);
}

// 只要按下鼠标按键，
// 每次移动鼠标时都会调用此函数。
function dragObject(event:MouseEvent):void
{
    // 将拖动的对象移到光标的位置，从而保持
    // 光标的位置和拖动的对象的位置
    // 之间的偏移量。
    draggedObject.x = event.stageX - offsetX;
    draggedObject.y = event.stageY - offsetY;

    // 指示 Flash Player 在此事件后刷新屏幕。
    event.updateAfterEvent();
}

circle.addEventListener(MouseEvent.MOUSE_DOWN, startDragging);
circle.addEventListener(MouseEvent.MOUSE_UP, stopDragging);

square.addEventListener(MouseEvent.MOUSE_DOWN, startDragging);
square.addEventListener(MouseEvent.MOUSE_UP, stopDragging);
```

要进一步扩展这种效果，如在几副纸牌（或几组标记）之间移动纸牌（或标记）的游戏中，您可以在"拿出"拖动对象时将拖动对象添加到舞台的显示列表中，然后在"放入"拖动对象时（通过松开鼠标按键）将拖动对象添加到另一个显示列表中（如"那副纸牌"或"那组标记"）。

最后，要增强效果，您可以在单击显示对象时（开始拖动显示对象时）对显示对象应用投影滤镜，然后在松开对象时删除投影。有关在 ActionScript 中使用投影滤镜和其它显示对象滤镜的详细信息，请参阅第 403 页的第 15 章"过滤显示对象"。

## 平移和滚动显示对象

如果显示对象太大，不能在要显示它的区域中完全显示出来，则可以使用 scrollRect 属性定义显示对象的可查看区域。此外，通过更改 scrollRect 属性响应用户输入，可以使内容左右平移或上下滚动。

scrollRect 属性是 Rectangle 类的实例， Rectangle 类包括将矩形区域定义为单个对象所需的有关值。最初定义显示对象的可查看区域时，请创建一个新的 Rectangle 实例并为该实例分配显示对象的 scrollRect 属性。以后进行滚动或平移时，可以将 scrollRect 属性读入单独的 Rectangle 变量，然后更改所需的属性（例如，更改 Rectangle 实例的 x 属性进行平移，或更改 y 属性进行滚动）。然后将该 Rectangle 实例重新分配给 scrollRect 属性，将更改的值通知显示对象。

例如，下面的代码定义了名为 bigText 的 TextField 对象的可查看区域，该对象因太高而不能适合 SWF 文件的边界。单击名为 up 和 down 的两个按钮时，它们调用的函数通过修改 scrollRect Rectangle 实例的 y 属性而使 TextField 对象的内容向上或向下滚动。

```
import flash.events.MouseEvent;
import flash.geom.Rectangle;

// 定义 TextField 实例的最初可查看区域:
// 左: 0，上: 0，宽度: TextField 的宽度，高度: 350 个像素。
bigText.scrollRect = new Rectangle(0, 0, bigText.width, 350);

// 将 TextField 作为位图缓存以提高性能。
bigText.cacheAsBitmap = true;

// 单击 “向上”按钮时调用
function scrollUp(event:MouseEvent):void
{
  // 访问当前滚动矩形。
  var rect:Rectangle = bigText.scrollRect;
  // 将矩形的 y 值减小 20，
  // 从而使矩形有效下移 20 个像素。
  rect.y -= 20;
  // 将矩形重新分配给 TextField 以 “应用”更改。
  bigText.scrollRect = rect;
}

// 单击 “向下”按钮时调用
function scrollDown(event:MouseEvent):void
{
  // 访问当前滚动矩形。
  var rect:Rectangle = bigText.scrollRect;
  // 将矩形的 y 值增加 20，
  // 从而使矩形有效上移 20 个像素。
  rect.y += 20;
  // 将矩形重新分配给 TextField 以 “应用”更改。
  bigText.scrollRect = rect;
}
```

```
up.addEventListener(MouseEvent.CLICK, scrollUp);
down.addEventListener(MouseEvent.CLICK, scrollDown);
```

正如本示例所示，使用显示对象的 scrollRect 属性时，最好指定 Flash Player 应使用 cacheAsBitmap 属性将显示对象的内容作为位图来缓存。如果这样做了，每次滚动显示对象时， Flash Player 就不必重绘显示对象的整个内容，而可以改为使用缓存的位图将所需部分直接呈现到屏幕上。有关详细信息，请参阅第 349 页的 “缓存显示对象”。

## 处理大小和缩放对象

您可以采用两种方法来测量和处理显示对象的大小：使用尺寸属性 （width 和 height）或缩放属性 （scaleX 和 scaleY）。

每个显示对象都有 width 属性和 height 属性，它们最初设置为对象的大小，以像素为单位。您可以通过读取这些属性的值来确定显示对象的大小。还可以指定新值来更改对象的大小，如下所示：

```
// 调整显示对象的大小。
square.width = 420;
square.height = 420;

// 确定圆显示对象的半径。
var radius:Number = circle.width / 2;
```

更改显示对象的 height 或 width 会导致缩放对象，这意味着对象内容经过伸展或挤压以适合新区域的大小。如果显示对象仅包含矢量形状，将按新缩放比例重绘这些形状，而品质不变。此时将缩放显示对象中的所有位图图形元素，而不是重绘。例如，缩放图形时，如果数码照片的宽度和高度增加后超出图像中像素信息的实际大小，数码照片将被像素化，使数码照片显示带有锯齿。

当更改显示对象的 width 或 height 属性时， Flash Player 还会更新对象的 scaleX 和 scaleY 属性。这些属性表示显示对象与其原始大小相比的相对大小。 scaleX 和 scaleY 属性使用小数（十进制）值来表示百分比。例如，如果某个显示对象的 width 已更改，其宽度是原始大小的一半，则该对象的 scaleX 属性的值为 .5，表示 50%。如果其高度加倍，则其 scaleY 属性的值为 2，表示 200%。

```
// 圆是一个宽度和高度均为 150 个像素的显示对象。
// 按照原始大小，scaleX 和 scaleY 均为 1 (100%)。
trace(circle.scaleX);// 输出: 1
trace(circle.scaleY);// 输出: 1

// 当更改 width 和 height 属性时，
// Flash Player 会相应更改 scaleX 和 scaleY 属性。
circle.width = 100;
circle.height = 75;
trace(circle.scaleX);// 输出: 0.6622516556291391
trace(circle.scaleY);// 输出: 0.4966887417218543
```

此时，大小更改不成比例。换句话说，如果更改一个正方形的 height 但不更改其 width，则其边长不再相同，它将是一个矩形而不是一个正方形。如果要更改显示对象的相对大小，则可以通过设置 scaleX 和 scaleY 属性的值来调整该对象的大小，另一种方法是设置 width 或 height 属性。例如，下面的代码将更改名为 square 的显示对象的 width，然后更改垂直缩放 (scaleY) 以匹配水平缩放，所以正方形的大小成比例。

```
// 直接更改宽度。
square.width = 150;

// 更改垂直缩放以匹配水平缩放，
// 使大小成比例。
square.scaleY = square.scaleX;
```

## 控制缩放时的扭曲

通常，缩放显示对象（例如水平伸展）时，引起的扭曲在整个对象上是均匀分布的，所以各部分的伸展量是相同的。对于图形和设计元素，这可能是您所希望的结果。但是，有时更希望能控制显示对象的某些部分伸展、某些部分保持不变。这种情况的一个常见示例是有圆角的矩形按钮。进行正常缩放时，按钮的角将伸展，从而使角半径随按钮大小的调整而改变。



但在这种情况下，最好对缩放进行控制，即能够指定应缩放的某些区域（直边和中间）和不应缩放的区域（角），以便在缩放后不会出现可见的扭曲。



可以使用 9 切片缩放 (Scale-9) 来创建在其中控制如何缩放对象的显示对象。使用 9 切片缩放时，显示对象被分成 9 个单独的矩形（一个 3 x 3 的网格，就像一个“井”字）。矩形的大小不必一定相同，您可以指定放置网格线的位置。缩放显示对象时，四个角矩形中的任何内容（如按钮的圆角）不伸展也不压缩。上中矩形和下中矩形将进行水平缩放，但不进行垂直缩放，而左中矩形和右中矩形将进行垂直缩放，但不进行水平缩放。中心矩形既进行水平缩放又进行垂直缩放。



请记住，如果要创建显示对象且希望某些内容从不缩放，只需要通过放置 9 切片缩放网格的划分线来确保有关内容完全放在其中一个角矩形中即可。

在 ActionScript 中，如果为显示对象的 `scale9Grid` 属性设置一个值，就会打开对象的 9 切片缩放并定义对象的 Scale-9 网格中矩形的大小。可以使用 Rectangle 类的实例作为 `scale9Grid` 属性的值，如下所示：

```
myButton.scale9Grid = new Rectangle(32, 27, 71, 64);
```

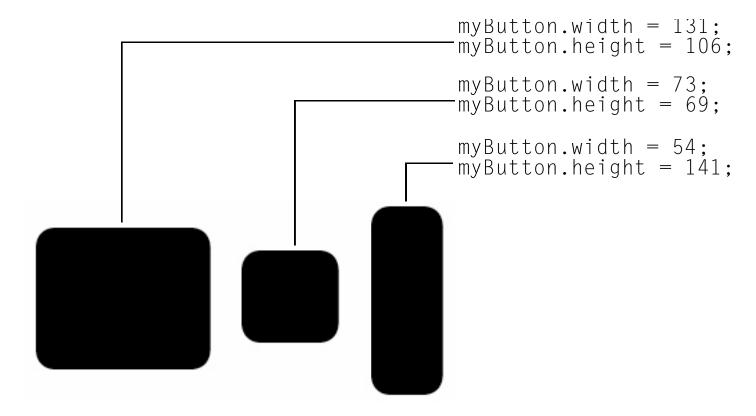
Rectangle 构造函数的四个参数是 x 坐标、y 坐标、width 和 height。本示例中，矩形的左上角放在名为 `myButton` 的显示对象上的点 x: 32，y: 27 处。矩形宽 71 个像素，高 65 个像素（因此其右边位于显示对象上 x 坐标为 103 的位置，其下边位于显示对象上 y 坐标为 92 的位置）。



包含在由 Rectangle 实例定义的区域中的实际区域表示 Scale-9 网格的中心矩形。其它矩形是由 Flash Player 通过扩展 Rectangle 实例的各边计算出来的，如下所示：



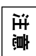
在本例中，当按钮放大或缩小时，圆角不拉伸也不压缩，但其它区域将通过调整来适应缩放。




## 缓存显示对象

如果 Flash 中的设计尺寸增大，无论创建的是应用程序还是复杂的脚本动画，都需要考虑性能和优化。如果内容保持为静态（如矩形 Shape 实例）， Flash 不会优化内容。因此，更改矩形的位置时， Flash 将重绘整个 Shape 实例。

可以通过缓存指定的显示对象来提高 SWF 文件的性能。显示对象是一个 "表面"，实际上是位图版本的实例矢量数据，矢量数据是 SWF 文件中不需要有太多更改的一种数据。因此，打开缓存的实例不会随 SWF 文件的播放而不断地重绘，这样便可快速呈现 SWF 文件。



注意：可以更新矢量数据，这时将重新创建表面。因此，缓存在表面中的矢量数据不需要在整个 SWF 文件中保持一样。

将显示对象的 `cacheAsBitmap` 属性设置为 `true` 会使显示对象缓存其自身的位图表示。Flash 为该实例创建一个 surface 对象，该对象是一个缓存的位图，而不是矢量数据。如果要更改显示对象的边界，则重新创建表面而不是调整其大小。表面可以嵌套在其它表面之内。子表面会将其位图复制到它的父表面上。有关详细信息，请参阅第 351 页的 "启用位图缓存"。

DisplayObject 类的 `opaqueBackground` 属性和 `scrollRect` 属性与使用 `cacheAsBitmap` 属性的位图缓存有关。尽管这三个属性彼此互相独立，但是，当对象缓存为位图时，`opaqueBackground` 和 `scrollRect` 属性的作用最佳，只有将 `cacheAsBitmap` 设置为 `true` 时，才能看到 `opaqueBackground` 和 `scrollRect` 属性带来的性能优势。有关滚动显示对象内容的详细信息，请参阅第 345 页的 "平移和滚动显示对象"。有关设置不透明背景的详细信息，请参阅第 351 页的 "设置不透明背景颜色"。

有关 Alpha 通道遮罩（要求将 `cacheAsBitmap` 属性设置为 `true`）的信息，请参阅第 357 页的 "Alpha 通道遮罩"。

## 何时启用缓存

对显示对象启用缓存可创建表面，表面具有助于更快地呈现复杂的矢量动画等优点。有几种情形需要启用缓存。可能您总是希望通过启用缓存来提高 SWF 文件的性能；但是，某些情况下启用缓存并不能提高性能，甚至还会降低性能。本部分介绍在哪些情况下应使用缓存，以及何时使用常规显示对象。

缓存数据的总体性能取决于实例矢量数据的复杂程度、要更改的数据量，以及是否设置了 `opaqueBackground` 属性。如果要更改的区域较小，则使用表面和使用矢量数据的差异微乎其微。在部署应用程序之前您可能需要实际测试一下这两种情况。

### 何时使用位图缓存

在以下典型情况下，启用位图缓存可能会带来明显的好处。

- 复杂的背景图像：应用程序包含由矢量数据组成的细节丰富且背景复杂的图像 （可能是应用了跟踪位图命令的图像，也可能是在 Adobe Illustrator® 中创建的图片）。您可能会在背景上设计动画人物，这会降低动画的速度，因为背景需要持续地重新生成矢量数据。要提高性能，可以将背景显示对象的 `opaqueBackground` 属性设置为 `true`。背景将呈现为位图，可以迅速地重绘，所以动画的播放速度比较快。
- 滚动文本字段：应用程序在滚动文本字段中显示大量的文本。可以将文本字段放置在您设置为可滚动的具有滚动框 （使用 `scrollRect` 属性）的显示对象中。这可以使指定的实例进行快速像素滚动。当用户滚动显示对象实例时，Flash 通过将滚动的像素向上移来生成新的看得见的区域，而不是重新生成整个文本字段。
- 窗口排列秩序：应用程序具有秩序复杂的重叠窗口。每个窗口都可以打开或关闭（例如，Web 浏览器窗口） 。如果将每个窗口标记为一个表面 （将 `cacheAsBitmap` 属性设置为 `true`），则各个窗口将隔离开来进行缓存。用户可以拖动窗口使其互相重叠，每个窗口并不重新生成矢量内容。
- Alpha 通道遮罩：当使用 Alpha 通道遮罩时，必须将 `cacheAsBitmap` 属性设置为 `true`。有关详细信息，请参阅第 357 页的 “Alpha 通道遮罩”。

所有这些情况下，启用位图缓存后都通过优化矢量图来提高应用程序的响应能力和互动性。

此外，只要对显示对象应用滤镜， Flash Player 就会将 `cacheAsBitmap` 自动设置为 `true`，即使已明确将其设置为 `false` 也是如此。如果清除了显示对象的所有滤镜，则 `cacheAsBitmap` 属性会返回最后设置的值。

### 何时避免使用位图缓存

滥用此功能对 SWF 文件可能会有负面影响。使用位图缓存时，请记住下面的准则：

- 不要过度使用表面 （启用了缓存的显示对象） 。每个表面使用的内存都比常规显示对象多，这意味着只在需要提高呈现性能时才启用表面。

  缓存的位图使用的内存比常规显示对象多很多。例如，如果舞台上 Sprite 实例的大小为 250 x 250 个像素，缓存它时可能会使用 250 KB 的内存，如果它是常规 （未缓存的） Sprite 实例，则使用 1 KB 的内存。
- 避免放大缓存的表面。如果过度使用位图缓存，尤其是放大内容时，将使用大量的内存（请参阅上一段落）。
- 将表面用于通常为静态 （非动画）的显示对象实例。可以拖动或移动实例，但实例内容不应为动画或者有太多的变化。（动画或变化的内容更可能包含在包含动画的 MovieClip 实例或 Video 实例中。）例如，如果旋转或变形某一实例，实例在表面和矢量数据之间会有所改变，这种情况难于处理，对 SWF 文件会产生负面影响。
- 如果将表面和矢量数据混在一起，则会增加 Flash Player （有时还有计算机）需要处理的工作量。尽可能将表面归为一组 — 例如，创建窗口应用程序时。

## 启用位图缓存

要为显示对象启用位图缓存，请将它的 `cacheAsBitmap` 属性设置为 `true`：

```
mySprite.cacheAsBitmap = true;
```

将 `cacheAsBitmap` 属性设置为 `true` 后，您可能会注意到，显示对象的像素会自动与整个坐标对齐。测试 SWF 文件时，您还会注意到，在复杂矢量图像上执行的任何动画的呈现速度都快得多。

即便将 `cacheAsBitmap` 已设置为 `true`，如果出现以下一种或多种情况，也不会创建表面（缓存的位图）：

- 位图高度或宽度超过 2880 个像素。
- 位图分配不成功 （由于内存不足而出现的错误）。

## 设置不透明背景颜色

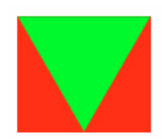
可以为显示对象设置不透明背景。例如，如果 SWF 的背景中包含复杂的矢量图片，则可以将 `opaqueBackground` 属性设置为指定的颜色 （通常与舞台颜色相同）。将颜色指定为一个数字 （通常为十六进制的颜色值）。然后将背景视为一个位图，这有助于优化性能。

当将 `cacheAsBitmap` 设置为 `true` 并将 `opaqueBackground` 属性设置为指定的颜色时，`opaqueBackground` 属性可以使内部位图不透明而加快呈现速度。如果不将 `cacheAsBitmap` 设置为 `true`，`opaqueBackground` 属性将在显示对象的背景中添加一个不透明的矢量正方形形状。它不会自动创建位图。

下面的示例说明了如何设置显示对象的背景以优化性能。

```
myShape.cacheAsBitmap = true;
myShape.opaqueBackground = 0xFF0000;
```

在本例中，将名为 `myShape` 的 Shape 的背景颜色设置为红色 (`0xFF0000`)。假定 Shape 实例在白色背景的舞台上包含一个绿色三角形绘图，这将在 Shape 实例的边框（完全包含 Shape 的矩形）内显示一个绿色三角形，且空白区域为红色。



当然，如果此代码用于纯红色背景的舞台，则更合理。在其它颜色的背景上，则改为指定该颜色。例如，在白色背景的 SWF 中，`opaqueBackground` 属性最适合设置为 `0xFFFFFF` 或纯白色。

## 应用混合模式

混合模式涉及将一个图像（基图像）的颜色与另一个图像（混合图像）的颜色进行组合来生成第三个图像，结果图像是实际在屏幕上显示的图像。图像中的每个像素值都与另一个图像的对应像素值一起处理的，以便为结果图像中的相同位置生成像素值。

每个显示对象都有 `blendMode` 属性，可以将其设置为下列混合模式之一。以下是在 BlendMode 类中定义的常量。此外，还可以使用 String 值 （在括号中），这些值是常量的实际值。

- `BlendMode.ADD` (`"add"`)：通常用于创建两个图像之间的动画变亮模糊效果。
- `BlendMode.ALPHA` (`"alpha"`)：通常用于在背景上应用前景的透明度。
- `BlendMode.DARKEN` (`"darken"`)：通常用于重叠类型。
- `BlendMode.DIFFERENCE` (`"difference"`)：通常用于创建更多变动的颜色。
- `BlendMode.ERASE` (`"erase"`)：通常用于使用前景 Alpha 剪掉 （擦除）背景的一部分。
- `BlendMode.HARDLIGHT` (`"hardlight"`)：通常用于创建阴影效果。
- `BlendMode.INVERT` (`"invert"`)：用于反转背景。
- `BlendMode.LAYER` (`"layer"`)：用于强制为特定显示对象的预构成创建临时缓冲区。
- `BlendMode.LIGHTEN` (`"lighten"`)：通常用于重叠类型。
- `BlendMode.MULTIPLY` (`"multiply"`)：通常用于创建阴影和深度效果。
- `BlendMode.NORMAL` (`"normal"`)：用于指定混合图像的像素值覆盖基本图像的像素值。
- `BlendMode.OVERLAY` (`"overlay"`)：通常用于创建阴影效果。
- `BlendMode.SCREEN` (`"screen"`)：通常用于创建亮点和镜头眩光。
- `BlendMode.SUBTRACT` (`"subtract"`)：通常用于创建两个图像之间的动画变暗模糊效果。

## 调整 DisplayObject 颜色

可以使用 ColorTransform 类的方法 (flash.geom.ColorTransform) 来调整显示对象的颜色。每个显示对象都有 transform 属性（它是 Transform 类的实例），还包含有关应用到显示对象的各种变形的信息（如旋转、缩放或位置的更改等）。除了有关几何变形的信息之外，Transform 类还包括 colorTransform 属性，它是 ColorTransform 类的实例，并提供访问来对显示对象进行颜色调整。要访问显示对象的颜色转换信息，可以使用如下代码：

```
var colorInfo:ColorTransform = myDisplayObject.transform.colorTransform;
```

创建 ColorTransform 实例后，可以通过读取其属性值来查明已应用了哪些颜色转换，也可以通过设置这些值来更改显示对象的颜色。要在进行任何更改后更新显示对象，必须将 ColorTransform 实例重新分配给 transform.colorTransform 属性。

```
var colorInfo:ColorTransform = my DisplayObject.transform.colorTransform;

// 此处进行某些颜色转换。

// 提交更改。
myDisplayObject.transform.colorTransform = colorInfo;
```

## 使用代码设置颜色值

ColorTransform 类的 color 属性可用于为显示对象分配具体的红、绿、蓝 (RGB) 颜色值。在下面的示例中，当用户单击名为 blueBtn 的按钮时，将使用 color 属性将名为 square 的显示对象的颜色更改为蓝色：

```
// square 是舞台上的一个显示对象。
// blueBtn、redBtn、greenBtn 和 blackBtn 是舞台上的按钮。

import flash.events.MouseEvent;
import flash.geom.ColorTransform;

// 访问与 square 关联的 ColorTransform 实例。
var colorInfo:ColorTransform = square.transform.colorTransform;

// 单击 blueBtn 时会调用此函数。
function makeBlue(event:MouseEvent):void
{
    // 设置 ColorTransform 对象的颜色。
    colorInfo.color = 0x003399;
    // 将更改应用于显示对象
    square.transform.colorTransform = colorInfo;
}

blueBtn.addEventListener(MouseEvent.CLICK, makeBlue);
```

请注意，使用 color 属性更改显示对象的颜色时，将会完全更改整个对象的颜色，无论该对象以前是否有多种颜色。例如，如果某个显示对象包含一个顶部有黑色文本的绿色圆，将该对象的关联 ColorTransform 实例的 color 属性设置为红色阴影时，会使整个对象 （圆和文本）变为红色 （因此无法再将文本与对象的其余部分区分开来）。

## 使用代码更改颜色和亮度效果

假设显示对象有多种颜色 （例如，数码照片），但是您不想完全重新调整对象的颜色，只想根据现有颜色来调整显示对象的颜色。这种情况下，ColorTransform 类包括一组可用于进行此类调整的乘数属性和偏移属性。乘数属性的名分别为 redMultiplier、greenMultiplier、blueMultiplier 和 alphaMultiplier，它们的作用像彩色照片滤镜（或彩色太阳镜）一样，可以增强或削弱显示对象上的某些颜色。偏移属性（redOffset、greenOffset、blueOffset 和 alphaOffset）可用于额外增加对象上某种颜色的值，或用于指定特定颜色可以具有的最小值。

在 “属性”检查器上的 “颜色”弹出菜单中选择 “高级”时，这些乘数和偏移属性与 Flash 创作工具中影片剪辑元件可用的高级颜色设置相同。

下面的代码加载一个 JPEG 图像并为其应用颜色转换，当鼠标指针沿 x 轴和 y 轴移动时，将调整红色和绿色通道值。在本例中，由于未指定偏移值，因此屏幕上显示的每个颜色通道的颜色值将表示图像中原始颜色值的一个百分比，这意味着任何给定像素上显示的大部分红色或绿色都是该像素上红色或绿色的原始效果。

```
import flash.display.Loader;
import flash.events.MouseEvent;
import flash.geom.Transform;
import flash.geom.ColorTransform;
import flash.net.URLRequest;

// 将图像加载到舞台上。
var loader:Loader = new Loader();
var url:URLRequest = new URLRequest("http://www.helpexamples.com/flash/
  images/image1.jpg");
loader.load(url);
this.addChild(loader);

// 当鼠标移过加载的图像时会调用此函数。
function adjustColor(event:MouseEvent):void
{
  // 访问 Loader 的 ColorTransform 对象 （包含图像）
  var colorTransformer:ColorTransform = loader.transform.colorTransform;

  // 根据鼠标位置设置红色和绿色乘数。
  // 红色值的范围从 0% （无红色）（当光标位于左侧时）
  // 到 100% 红色 （正常图像外观）（当光标位于右侧时）。
  // 这同样适用于绿色通道，不同的是它由
  // y 轴中鼠标的位置控制。
```

```
    colorTransformer.redMultiplier = (loader.mouseX / loader.width) * 1;
    colorTransformer.greenMultiplier = (loader.mouseY / loader.height) * 1;

    // 将更改应用到显示对象。
    loader.transform.colorTransform = colorTransformer;
}

loader.addEventListener(MouseEvent.MOUSE_MOVE, adjustColor);
```

## 旋转对象

使用 rotation 属性可以旋转显示对象。可以通过读取此值来了解是否旋转了某个对象，如果要旋转该对象，可以将此属性设置为一个数字 （以度为单位），表示要应用于该对象的旋转量。例如，下面的代码行将名为 square 的对象旋转 45 度 （一整周旋转的 1/8）：

```
square.rotation = 45;
```

或者，还可以使用转换矩阵来旋转显示对象，在第 371 页的第 13 章 "处理几何结构"中予以介绍。

## 淡化对象

可以通过控制显示对象的透明度来使显示对象部分透明 （或完全透明），也可以通过更改透明度来使对象淡入或淡出。DisplayObject 类的 alpha 属性用于定义显示对象的透明度（更确切地说是不透明度）。可以将 alpha 属性设置为介于 0 和 1 之间的任何值，其中 0 表示完全透明， 1 表示完全不透明。例如，当使用鼠标单击名为 myBall 的对象时，下面的代码行将使该对象变得部分 (50%) 透明：

```
function fadeBall(event:MouseEvent):void
{
    myBall.alpha = .5;
}
myBall.addEventListener(MouseEvent.CLICK, fadeBall);
```

还可以使用通过 ColorTransform 类提供的颜色调整来更改显示对象的透明度。有关详细信息，请参阅第 353 页的 "调整 DisplayObject 颜色"。

# 遮罩显示对象

可以通过将一个显示对象用作遮罩来创建一个孔洞，透过该孔洞使另一个显示对象的内容可见。

## 定义遮罩

要指明一个显示对象将是另一个显示对象的遮罩，请将遮罩对象设置为被遮罩的显示对象的 mask 属性：

```
// 使对象 maskSprite 成为对象 mySprite 的遮罩。
mySprite.mask = maskSprite;
```

被遮罩的显示对象显示在用作遮罩的显示对象的全部不透明区域之内。例如，下面的代码将创建一个包含 100 x 100 个像素的红色正方形的 Shape 实例和一个包含半径为 25 个像素的蓝色圆的 Sprite 实例。单击圆时，它被设置为正方形的遮罩，所以显示的正方形部分只是由圆完整部分覆盖的那一部分。换句话说，只有红色圆可见。

```
// 以下代码假设它正在显示对象容器
//（如 MovieClip 或 Sprite 实例）中运行。

import flash.display.Shape;

// 绘制正方形并将其添加到显示列表中。
var square:Shape = new Shape();
square.graphics.lineStyle(1, 0x000000);
square.graphics.beginFill(0xff0000);
square.graphics.drawRect(0, 0, 100, 100);
square.graphics.endFill();
this.addChild(square);

// 绘制圆并将其添加到显示列表中。
var circle:Sprite = new Sprite();
circle.graphics.lineStyle(1, 0x000000);
circle.graphics.beginFill(0x0000ff);
circle.graphics.drawCircle(25, 25, 25);
circle.graphics.endFill();
this.addChild(circle);

function maskSquare(event:MouseEvent):void
{
  square.mask = circle;
  circle.removeEventListener(MouseEvent.CLICK, maskSquare);
}

circle.addEventListener(MouseEvent.CLICK, maskSquare);
```

用作遮罩的显示对象可拖动、设置动画，并可动态调整大小，可以在单个遮罩内使用单独的形状。遮罩显示对象不必一定需要添加到显示列表中。但是，如果希望在缩放舞台时也缩放遮罩对象，或者如果希望支持用户与遮罩对象的交互（如用户控制的拖动和调整大小），则必须将遮罩对象添加到显示列表中。遮罩对象已添加到显示列表时，显示对象的实际 z 索引（从前到后顺序）并不重要。（除了显示为遮罩对象外，遮罩对象将不会出现在屏幕上。）如果遮罩对象是包含多个帧的一个 MovieClip 实例，则遮罩对象会沿其时间轴播放所有帧，如果没有用作遮罩对象，也会出现同样的情况。通过将 mask 属性设置为 null 可以删除遮罩：

```
// 删除 mySprite 中的遮罩
mySprite.mask = null;
```

不能使用一个遮罩对象来遮罩另一个遮罩对象。不能设置遮罩显示对象的 alpha 属性。只有填充可用于作为遮罩的显示对象中；笔触都会被忽略。

## 关于遮蔽设备字体

您可以使用显示对象遮罩用设备字体设置的文本。当使用显示对象遮罩用设备字体设置的文本时，遮罩的矩形边框会用作遮罩形状。也就是说，如果为设备字体文本创建了非矩形的显示对象遮罩，则 SWF 文件中显示的遮罩将是遮罩的矩形边框的形状，而不是遮罩本身的形状。

## Alpha 通道遮罩

如果遮罩显示对象和被遮罩的显示对象都使用位图缓存，则支持 Alpha 通道遮罩，如下所示：

```
// maskShape 是一个包括渐变填充的 Shape 实例。
mySprite.cacheAsBitmap = true;
maskShape.cacheAsBitmap = true;
mySprite.mask = maskShape;
```

例如， Alpha 通道遮罩的一个应用是对遮罩对象使用应用于被遮罩显示对象之外的滤镜。

在下面的示例中，将一个外部图像文件加载到舞台上。该图像（更确切地说，是加载图像的 Loader 实例）将是被遮罩的显示对象。渐变椭圆（中心为纯黑色，边缘淡变为透明）绘制在图像上；这就是 Alpha 遮罩。两个显示对象都打开了位图缓存。椭圆设置为图像的遮罩，然后使其可拖动。

```
// 以下代码假设它正在显示对象容器
//（如 MovieClip 或 Sprite 实例）中运行。

import flash.display.GradientType;
import flash.display.Loader;
import flash.display.Sprite;
import flash.geom.Matrix;
import flash.net.URLRequest;
```

```
// 加载图像并将其添加到显示列表中。
var loader:Loader = new Loader();
var url:URLRequest = new URLRequest("http://www.helpexamples.com/flash/
  images/image1.jpg");
loader.load(url);
this.addChild(loader);

// 创建 Sprite。
var oval:Sprite = new Sprite();
// 绘制渐变椭圆。
var colors:Array = [0x000000, 0x000000];
var alphas:Array = [1, 0];
var ratios:Array = [0, 255];
var matrix:Matrix = new Matrix();
matrix.createGradientBox(200, 100, 0, -100, -50);
oval.graphics.beginGradientFill(GradientType.RADIAL,
                    colors,
                    alphas,
                    ratios,
                    matrix);
oval.graphics.drawEllipse(-100, -50, 200, 100);
oval.graphics.endFill();
// 将 Sprite 添加到显示列表中
this.addChild(oval);

// 对于两个显示对象都设置 cacheAsBitmap = true。
loader.cacheAsBitmap = true;
oval.cacheAsBitmap = true;
// 将椭圆设置为加载器（及其子级，即加载的图像）的遮罩
loader.mask = oval;

// 使椭圆可拖动。
oval.startDrag(true);
```

# 对象动画

动画是使内容移动或者使内容随时间发生变化的过程。脚本动画是视频游戏的基础部分，通常用于将优美、有用的交互线索添加到其它应用程序中。

脚本动画的基本概念是变化一定要发生，而且变化一定要分时间逐步完成。使用常见的循环语句，可很容易在 ActionScript 中使内容重复。但是，在更新显示之前，循环将遍历其所有迭代。要创建脚本动画，需要编写 ActionScript，它随时间重复执行某个动作，每次运行时还更新屏幕。

例如，假设要创建一个简单的动画，如使球沿着屏幕运动。ActionScript 包括一个允许您跟踪时间和相应更新屏幕的简单机制，这意味着您可以编写代码，每次让球移动一点点，直到球到达目标为止。每次移动后，屏幕都会更新，从而使跨舞台的运动在查看器中可见。

从实际观点来看，让脚本动画与 SWF 文件的帧速率同步（换句话说，每次新帧显示时都设计一个动画变化）才有意义，因为这是 Flash Player 更新屏幕的速度。每个显示对象都有 enterFrame 事件，它根据 SWF 文件的帧速率来调度，即每帧一个事件。创建脚本动画的大多数开发人员都使用 enterFrame 事件作为一种方法来创建随时间重复的动作。可以编写代码以侦听 enterFrame 事件，每一帧都让动画球移动一定的量，当屏幕更新时（每一帧），将会在新位置重新绘制该球，从而产生了运动。

**注意** 另一种随时间重复执行某个动作的方法是使用 Timer 类。每次过了指定的时间时， Timer 实例都会触发事件通知。可以编写通过处理 Timer 类的 timer 事件来执行动画的代码，将时间间隔设置为一个很小的间隔（几分之几秒）。有关使用 Timer 类的详细信息，请参阅第 164 页的 “控制时间间隔”。

在下面的示例中，将在舞台上创建一个名为 circle 的圆 Sprite 实例。当用户单击圆时，脚本动画序列开始，从而使 circle 淡化 （其 alpha 属性值减少），直到完全透明：

```
import flash.display.Sprite;
import flash.events.Event;
import flash.events.MouseEvent;

// 绘制圆并将其添加到显示列表中
var circle:Sprite = new Sprite();
circle.graphics.beginFill(0x990000);
circle.graphics.drawCircle(50, 50, 50);
circle.graphics.endFill();
addChild(circle);

// 此动画开始后，每一帧都会调用此函数。
// 此函数进行的更改
// （每一帧都会更新屏幕）将导致产生动画效果。
function fadeCircle(event:Event):void
{
  circle.alpha -= .05;

  if (circle.alpha <= 0)
  {
    circle.removeEventListener(Event.ENTER_FRAME, fadeCircle);
  }
}

function startAnimation(event:MouseEvent):void
{
  circle.addEventListener(Event.ENTER_FRAME, fadeCircle);
}

circle.addEventListener(MouseEvent.CLICK, startAnimation);
```

当用户单击圆时，将函数 fadeCircle() 订阅为 enterFrame 事件的侦听器，这意味着每一帧都会开始调用一次该函数。通过更改 circle 的 alpha 属性，该函数会淡化圆，因此对于每个帧，圆的 alpha 一次减少 .05（5%），并且更新屏幕。最后，当 alpha 值为 0（circle 完全透明）时， fadeCircle() 函数作为事件侦听器将被删除，从而结束动画。

以上代码还可用来创建动画运动而不是淡化。通过用不同属性替换函数中表示 enterFrame 事件侦听器的 alpha，就可获得该属性的动画效果。例如，将以下行

```
circle.alpha -= .05;
```

更改为以下代码

```
circle.x += 5;
```

将获得 x 属性的动画效果，从而使圆在舞台上移到右侧。当到达需要的 x 坐标时，通过更改结束动画的条件就可结束动画 （即取消订阅 enterFrame 侦听器）。

# 动态加载显示内容

可以将下列任何外部显示资源加载到 ActionScript 3.0 应用程序中：

- 在 ActionScript 3.0 中创作的 SWF 文件 — 此文件可以是 Sprite、 MovieClip 或扩展 Sprite 的任何类。
- 图像文件 — 包括 JPG、 PNG 和 GIF 文件。
- AVM1 SWF 文件 — 在 ActionScript 1.0 或 2.0 中编写的 SWF 文件。

使用 Loader 类可以加载这些资源。

## 加载显示对象

Loader 对象用于将 SWF 文件和图形文件加载到应用程序中。 Loader 类是 DisplayObjectContainer 类的子类。 Loader 对象在其显示列表中只能包含一个子显示对象，该显示对象表示它加载的 SWF 或图形文件。如下面的代码所示，在显示列表中添加 Loader 对象时，还可以在加载后将加载的子显示对象添加到显示列表中：

```
var pictLdr:Loader = new Loader();
var pictURL:String = "banana.jpg"
var pictURLReq:URLRequest = new URLRequest(pictURL);
pictLdr.load(pictURLReq);
this.addChild(pictLdr);
```

加载 SWF 文件或图像后，即可将加载的显示对象移到另一个显示对象容器中，例如本示例中的 container DisplayObjectContainer 对象：

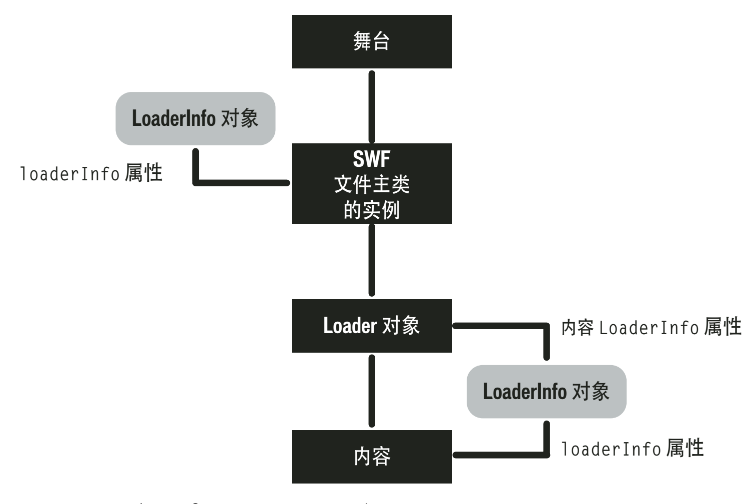
```
import flash.display.*;
import flash.net.URLRequest;
import flash.events.Event;
var container:Sprite = new Sprite();
```

```
addChild(container);
var pictLdr:Loader = new Loader();
var pictURL:String = "banana.jpg"
var pictURLReq:URLRequest = new URLRequest(pictURL);
pictLdr.load(pictURLReq);
pictLdr.contentLoaderInfo.addEventListener(Event.COMPLETE, imgLoaded);
function imgLoaded(event:Event):void
{
  container.addChild(pictLdr.content);
}
```

## 监视加载进度

文件开始加载后，就创建了 LoaderInfo 对象。LoaderInfo 对象用于提供加载进度、加载者和被加载者的 URL、媒体的字节总数及媒体的标称高度和宽度等信息。LoaderInfo 对象还调度用于监视加载进度的事件。

下图说明了 LoaderInfo 对象的不同用途 — 用于 SWF 文件的主类的实例、用于 Loader 对象以及用于由 Loader 对象加载的对象：




可以将 LoaderInfo 对象作为 Loader 对象和加载的显示对象的属性进行访问。加载一开始，就可以通过 Loader 对象的 `contentLoaderInfo` 属性访问 LoaderInfo 对象。显示对象完成加载后，也可以将 LoaderInfo 对象作为加载的显示对象的属性通过显示对象的 `loaderInfo` 属性进行访问。已加载显示对象的 `loaderInfo` 属性是指与 Loader 对象的 `contentLoaderInfo` 属性相同的 LoaderInfo 对象。换句话说， LoaderInfo 对象是加载的对象与加载它的 Loader 对象之间（加载者和被加载者之间）的共享对象。

要访问加载的内容的属性，需要在 LoaderInfo 对象中添加事件侦听器，如下面的代码所示:

```
import flash.display.Loader;
import flash.display.Sprite;
import flash.events.Event;

var ldr:Loader = new Loader();
var urlReq:URLRequest = new URLRequest("Circle.swf");
ldr.load(urlReq);
ldr.contentLoaderInfo.addEventListener(Event.COMPLETE, loaded);
addChild(ldr);

function loaded(event:Event):void
{
  var content:Sprite = event.target.content;
  content.scaleX = 2;
}
```

有关详细信息，请参阅第 267 页的第 10 章 “处理事件”。

## 指定加载上下文

通过 Loader 类的 `load()` 或 `loadBytes()` 方法将外部文件加载到 Flash Player 中时，可以选择性地指定 `context` 参数。此参数是一个 LoaderContext 对象。

LoaderContext 类包括三个属性，用于定义如何使用加载的内容的上下文：

- `checkPolicyFile`：仅当加载图像文件（不是 SWF 文件）时才会使用此属性。如果将此属性设置为 `true`，Loader 将检查跨域策略文件的原始服务器（请参阅第 656 页的 “Web 站点控制（跨域策略文件）”）。只有内容的来源域不是包含 Loader 对象的 SWF 文件所在的域时才需要此属性。如果服务器授予 Loader 域权限， Loader 域中 SWF 文件的 ActionScript 就可以访问加载图像中的数据；换句话说，可以使用 `BitmapData.draw()` 命令访问加载的图像中的数据。

  请注意，来自 Loader 对象所在域以外的其它域的 SWF 文件可以通过调用 `Security.allowDomain()` 来允许特定的域。

- `securityDomain`：仅当加载 SWF 文件（不是图像）时才会使用此属性。如果 SWF 文件所在的域与包含 Loader 对象的文件所在的域不同，则指定此属性。指定此选项时，Flash Player 将检查跨域策略文件是否存在，如果存在，来自跨策略文件中允许的域的 SWF 文件可以对加载的 SWF 内容执行跨脚本操作。可以将 `flash.system.SecurityDomain.currentDomain` 指定为此参数。

■ applicationDomain：仅当加载使用 ActionScript 3.0 编写的 SWF 文件（不是图像或使用 ActionScript 1.0 或 2.0 编写的 SWF 文件）时才会使用此属性。加载文件时，通过将 applicationDomain 参数设置为 flash.system.ApplicationDomain.currentDomain，可以指定将该文件包括在与 Loader 对象相同的应用程序域中。通过将加载的 SWF 文件放在同一个应用程序域中，可以直接访问它的类。如果要加载的 SWF 文件中包含嵌入的媒体，这会很有帮助，您可以通过其关联的类名访问嵌入的媒体。有关详细信息，请参阅第 603 页的 “使用 ApplicationDomain 类”。

下面的示例用于在加载来自另一个域的位图时检查跨域策略文件：

```
var context:LoaderContext = new LoaderContext();
context.checkPolicyFile = true;
var urlReq:URLRequest = new URLRequest("http://www.[your_domain_here].com/
  photo11.jpg");
var ldr:Loader = new Loader();
ldr.load(urlReq, context);
```

下面的示例用于从另一域加载 SWF 时检查跨域策略文件，以便将该文件放在与 Loader 对象相同的安全沙箱中。此外，该代码还将加载的 SWF 文件中的类添加到与 Loader 对象的类相同的应用程序域中：

```
var context:LoaderContext = new LoaderContext();
context.securityDomain = SecurityDomain.currentDomain;
context.applicationDomain = ApplicationDomain.currentDomain;
var urlReq:URLRequest = new URLRequest("http://www.[your_domain_here].com/
  library.swf");
var ldr:Loader = new Loader();
ldr.load(urlReq, context);
```

有关详细信息，请参阅 《ActionScript 3.0 语言和组件参考》中的 LoaderContext 类。

# 示例：SpriteArranger

SpriteArranger 示例应用程序是在 GeometricShapes 示例应用程序 （将单独进行介绍）的基础上构建的 （请参阅第 149 页的 “示例：GeometricShapes”）。

SpriteArranger 范例应用程序演示说明了处理显示对象的许多概念：

■ 扩展显示对象类

■ 在显示列表中添加对象

■ 分层显示对象和处理显示对象容器

■ 响应显示对象事件

■ 使用显示对象的属性和方法

要获取该范例的应用程序文件，请访问 www.adobe.com/go/learn_programmingAS3samples_flash_cn。可在文件夹 Examples/SpriteArranger 中找到 SpriteArranger 应用程序文件。该应用程序包含以下文件：

| 文件 | 说明 |
|---|---|
| SpriteArranger.mxml<br>或<br>SpriteArranger.fla | Flash 或 Flex 中的主应用程序文件 （分别为 FLA 和 MXML）。 |
| com/example/programmingas3/SpriteArranger/CircleSprite.as | 定义一种 Sprite 对象的类，该种对象在屏幕上呈现为圆。 |
| com/example/programmingas3/SpriteArranger/DrawingCanvas.as | 定义画布的类，画布是包含 GeometricSprite 对象的显示对象容器。 |
| com/example/programmingas3/SpriteArranger/SquareSprite.as | 定义一种 Sprite 对象的类，该对象在屏幕上呈现为正方形。 |
| com/example/programmingas3/SpriteArranger/TriangleSprite.as | 定义一种 Sprite 对象的类，该对象在屏幕上呈现为三角形。 |
| com/example/programmingas3/SpriteArranger/GeometricSprite.as | 扩展 Sprite 对象的类，用于定义屏幕形状。CircleSprite、 SquareSprite 和 TriangleSprite 都扩展此类。 |
| com/example/programmingas3/geometricshapes/IGeometricShape.as | 一个基接口，用于定义要由所有几何形状类实现的方法。 |
| com/example/programmingas3/geometricshapes/IPolygon.as | 一个接口，用于定义要由具有多条边的几何形状类实现的方法。 |
| com/example/programmingas3/geometricshapes/RegularPolygon.as | 一种边长相等且围绕形状的中心对称定位的几何形状。 |
| com/example/programmingas3/geometricshapes/Circle.as | 一种用于定义圆的几何形状。 |
| com/example/programmingas3/geometricshapes/EquilateralTriangle.as | RegularPolygon 的子类，用于定义所有边长相等的三角形。 |
| com/example/programmingas3/geometricshapes/Square.as | RegularPolygon 的子类，用于定义所有四条边相等的矩形。 |
| com/example/programmingas3/geometricshapes/GeometricShapeFactory.as | 包含 “工厂方法”的类，用于创建给定形状类型和大小的形状。 |

## 定义 SpriteArranger 类

用户可以使用 SpriteArranger 应用程序在屏幕 “画布” 上添加各种显示对象。

DrawingCanvas 类用于定义绘制区（一种显示对象容器），用户可以在其中添加屏幕形状。这些屏幕形状是 GeometricSprite 类的其中一个子类的实例。

### DrawingCanvas 类

DrawingCanvas 类扩展了 Sprite 类，此继承是在 DrawingCanvas 类声明中定义的，如下所示：

```
public class DrawingCanvas extends Sprite
```

Sprite 类是 DisplayObjectContainer 和 DisplayObject 类的子类，DrawingCanvas 类使用这些类的方法和属性。

`DrawingCanvas()` 构造函数方法设置 Rectangle 对象 `bounds`，它是以后在绘制画布轮廓时使用的属性。然后调用 `initCanvas()` 方法，如下所示：

```
this.bounds = new Rectangle(0, 0, w, h);
initCanvas(fillColor, lineColor);
```

如下面的示例所示，`initCanvas()` 方法用于定义 DrawingCanvas 对象的各种属性，这些属性作为参数传递给构造函数：

```
this.lineColor = lineColor;
this.fillColor = fillColor;
this.width = 500;
this.height = 200;
```

`initCanvas()` 方法随后调用 `drawBounds()` 方法，后者使用 DrawingCanvas 类的 `graphics` 属性绘制画布。`graphics` 属性继承自 Shape 类

```
this.graphics.clear();
this.graphics.lineStyle(1.0, this.lineColor, 1.0);
this.graphics.beginFill(this.fillColor, 1.0);
this.graphics.drawRect(bounds.left - 1,
             bounds.top - 1,
             bounds.width + 2,
             bounds.height + 2);
this.graphics.endFill();
```

DrawingCanvas 类的下列附加方法是根据用户与应用程序的交互进行调用的：

- `addShape()` 和 `describeChildren()` 方法，在第 366 页的 “在画布上添加显示对象” 中予以介绍。
- `moveToBack()`、`moveDown()`、`moveToFront()` 和 `moveUp()` 方法，在第 369 页的 “重新排列显示对象层” 中予以介绍。
- `onMouseUp()` 方法，在第 368 页的 “单击并拖动显示对象” 中予以介绍。

## GeometricSprite 类及其子类

用户在画布上可以添加的每个显示对象都是 GeometricSprite 类以下某个子类的实例：

- CircleSprite
- SquareSprite
- TriangleSprite

GeometricSprite 类扩展了 flash.display.Sprite 类：

```
public class GeometricSprite extends Sprite
```

GeometricSprite 类包括所有 GeometricSprite 对象所共有的一些属性。这些属性是根据传递到构造函数的参数，在该函数中设置的。例如：

```
this.size = size;
this.lineColor = lColor;
this.fillColor = fColor;
```

GeometricSprite 类的 `geometricShape` 属性用于定义 IGeometricShape 接口，该接口定义形状的数学属性，但不定义其可视属性。实现 IGeometricShape 接口的类是在 GeometricShapes 范例应用程序中定义的（请参阅第 149 页的 "示例：GeometricShapes"）。

GeometricSprite 类用于定义 `drawShape()` 方法，该方法在 GeometricSprite 的各子类的覆盖定义中已进一步精确定义。有关详细信息，请参阅后面的"在画布上添加显示对象"部分。

GeometricSprite 类还提供下列方法：

- `onMouseDown()` 和 `onMouseUp()` 方法，在第 368 页的"单击并拖动显示对象"中予以介绍。
- `showSelected()` 和 `hideSelected()` 方法，在第 368 页的 "单击并拖动显示对象"中予以介绍。

## 在画布上添加显示对象

当用户单击 "添加形状" 按钮时，应用程序将调用 DrawingCanvas 类的 `addShape()` 方法。它通过调用其中一个 GeometricSprite 子类的相应构造函数来实例化新的 GeometricSprite，如下面的示例所示：

```
public function addShape(shapeName:String, len:Number):void
{
  var newShape:GeometricSprite;
  switch (shapeName)
  {
    case "Triangle":
      newShape = new TriangleSprite(len);
      break;

    case "Square":
      newShape = new SquareSprite(len);
```

```
        break;

      case "Circle":
        newShape = new CircleSprite(len);
        break;
    }
    newShape.alpha = 0.8;
    this.addChild(newShape);
}
```

每个构造函数方法都调用 `drawShape()` 方法，该方法使用类（继承自 Sprite 类）的 `graphics` 属性来绘制相应的矢量图形。例如， CircleSprite 类的 `drawShape()` 方法包括下列代码：

```
this.graphics.clear();
this.graphics.lineStyle(1.0, this.lineColor, 1.0);
this.graphics.beginFill(this.fillColor, 1.0);
var radius:Number = this.size / 2;
this.graphics.drawCircle(radius, radius, radius);
```

`addShape()` 函数的倒数第二行用于设置显示对象（继承自 DisplayObject 类）的 `alpha` 属性，所以画布上添加的每个显示对象都有一点儿透明，这样用户就可看见它后面的对象。

`addChild()` 方法的最后一行用于在 DrawingCanvas 类实例的子级列表中添加新的显示对象，该实例已经在显示列表中。这样会使新的显示对象出现在舞台上。

应用程序界面上包括两个文本字段 `selectedSpriteTxt` 和 `outputTxt`。这些文本字段的文本属性由已添加到画布中或由用户选择的 GeometricSprite 对象的信息更新。 GeometricSprite 类通过覆盖 `toString()` 方法来处理这种信息报告任务，如下所示：

```
public override function toString():String
{
    return this.shapeType + “ of size “ + this.size + " at " + this.x + ", " +
    this.y;
}
```

`shapeType` 属性设置为每个 GeometricSprite 子类的构造函数方法中的相应值。例如，`toString()` 方法可能返回最近添加到 DrawingCanvas 实例中的 CircleSprite 实例的下列值：

```
Circle of size 50 at 0, 0
```

DrawingCanvas 类的 `describeChildren()` 方法通过使用 `numChildren` 属性（继承自 DisplayObjectContainer 类）设置 `for` 循环的限制，来遍历画布的子级列表。它会生成一个列出了每一子级的字符串，如下所示：

```
var desc:String = "";
var child:DisplayObject;
for (var i:int=0; i < this.numChildren; i++)
{
    child = this.getChildAt(i);
    desc += i + ": " + child + '\n';
}
```

生成的字符串用于设置 `outputTxt` 文本字段的 `text` 属性。

## 单击并拖动显示对象

当用户单击 GeometricSprite 实例时，应用程序将调用 `onMouseDown()` 事件处理函数。如下所示，此事件处理函数设置为侦听 GeometricSprite 类的构造函数中的鼠标按下事件：

```
this.addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);
```

`onMouseDown()` 方法随后调用 GeometricSprite 对象的 `showSelected()` 方法。如果是首次调用该对象的此方法，该方法将创建名为 `selectionIndicator` 的新 Shape 对象，并且使用 Shape 对象的 `graphics` 属性来绘制红色加亮矩形，如下所示：

```
this.selectionIndicator = new Shape();
this.selectionIndicator.graphics.lineStyle(1.0, 0xFF0000, 1.0);
this.selectionIndicator.graphics.drawRect(-1, -1, this.size + 1,
  this.size + 1);
this.addChild(this.selectionIndicator);
```

如果不是首次调用 `onMouseDown()` 方法，该方法仅设置 `selectionIndicator` 形状的 `visible` 属性（继承自 DisplayObject 类），如下所示：

```
this.selectionIndicator.visible = true;
```

`hideSelected()` 方法通过将其 `visible` 属性设置为 `false` 来隐藏以前所选对象的 `selectionIndicator` 形状。

`onMouseDown()` 事件处理函数方法还会调用 `startDrag()` 方法（继承自 Sprite 类），该方法包括下列代码：

```
var boundsRect:Rectangle = this.parent.getRect(this.parent);
boundsRect.width -= this.size;
boundsRect.height -= this.size;
this.startDrag(false, boundsRect);
```

这样，用户就可以在由 `boundsRect` 矩形设置的边界内在画布各处拖动选择的对象。

当用户松开鼠标按键时，将调度 `mouseUp` 事件。DrawingCanvas 的构造函数方法设置了下列事件侦听器：

```
this.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
```

此事件侦听器是针对 DrawingCanvas 对象设置的，而不是针对单个 GeometricSprite 对象设置的。这是因为当拖动 GeometricSprite 对象时，松开鼠标后它可能会在另一个显示对象（另一个 GeometricSprite 对象）之后结束前景中的显示对象会收到鼠标弹起事件，但用户正在拖动的显示对象则不会收到。在 DrawingCanvas 对象中添加侦听器可以确保始终处理该事件。

`onMouseUp()` 方法调用 GeometricSprite 对象的 `onMouseUp()` 方法，后者又调用 GeometricSprite 对象的 `stopDrag()` 方法。

## 重新排列显示对象层

应用程序用户界面上包括标有"后移"、"下移"、"上移"和"移到最前"的按钮。当用户单击其中一个按钮时，应用程序将调用 DrawingCanvas 类的相应方法：moveToBack()、moveDown()、moveUp() 或 moveToFront()。例如，moveToBack() 方法包括下列代码：

```
public function moveToBack(shape:GeometricSprite):void
{
    var index:int = this.getChildIndex(shape);
    if (index > 0)
    {
        this.setChildIndex(shape, 0);
    }
}
```

该方法使用 setChildIndex() 方法（继承自 DisplayObjectContainer 类）确定显示对象位置在 DrawingCanvas 实例 (this) 的子级列表中的索引位置 0。

moveDown() 方法的作用类似，不同的是它将显示对象在 DrawingCanvas 实例的子级列表中的索引位置减少 1：

```
public function moveDown(shape:GeometricSprite):void
{
    var index:int = this.getChildIndex(shape);
    if (index > 0)
    {
        this.setChildIndex(shape, index - 1);
    }
}
```

moveUp() 和 moveToFront() 方法的作用与 moveToBack() 和 moveDown() 方法类似。

第 13 章

# 处理几何结构

13

flash.geom 包中包含用于定义几何对象（如，点、矩形和转换矩阵）的类。您可使用这些类来定义在其它类中使用的对象的属性。

目录



## 几何学基础知识

### 处理几何学简介

很多人都可能会在学校努力学习几何学这门学科，过后又几乎忘记殆尽；但就算只了解一点这方面的知识，也可能会在 ActionScript 中派上大用场。

flash.geom 包中包含用于定义几何对象（如，点、矩形和转换矩阵）的类。这些类本身并不一定提供功能，但它们用于定义在其它类中使用的对象的属性。

所有几何类都基于以下概念：将屏幕上的位置表示为二维平面。可以将屏幕看作是具有水平 (x) 轴和垂直 (y) 轴的平面图形。屏幕上的任何位置（或 “点”）可以表示为 x 和 y 值对，即该位置的 “坐标”。

每个显示对象（包括舞台）具有其自己的“坐标空间”；实质上，这是其用于标绘子显示对象、图画等位置的图形。通常，“原点”（x 和 y 轴相交的位置，其坐标为 0, 0）位于显示对象的左上角。尽管这始终适用于舞台，但并不一定适用于任何其它显示对象。正如在标准二维坐标系中一样， x 轴上的值越往右越大，越往左越小；对于原点左侧的位置， x 坐标为负值。但是，与传统的坐标系相反，在 ActionScript 中，屏幕 y 轴上的值越往下越大，越往上越小（原点上面的 y 坐标为负值）。由于舞台左上角是其坐标空间的原点，因此，舞台上的任何对象的 x 坐标大于 0 并小于舞台宽度， y 坐标大于 0 并小于舞台高度。

可以使用 Point 类实例来表示坐标空间中的各个点。您可以创建一个 Rectangle 实例来表示坐标空间中的矩形区域。对于高级用户，可以使用 Matrix 实例将多个或复杂变形应用于显示对象。通过使用显示对象的属性，可以将很多简单变形（如旋转、位置以及缩放变化）直接应用于该对象。有关使用显示对象属性应用变形的详细信息，请参阅第 339 页的 “处理显示对象”。

## 常见几何学任务

您可能需要使用 ActionScript 中的几何类来完成以下任务：

- 计算两个点之间的距离
- 确定不同坐标空间中的点坐标
- 使用角度和距离移动显示对象
- 处理 Rectangle 实例：
  - 重新定位 Rectangle 实例
  - 调整 Rectangle 实例的大小
  - 确定 Rectangle 实例的组合大小或重叠区域
- 创建 Matrix 对象
- 使用 Matrix 对象将变形应用于显示对象

## 重要概念和术语

以下参考列表包含您将会在本章中遇到的重要术语：

- 笛卡尔坐标 (Cartesian coordinate)：通常，坐标采用一对数字的形式（如 5, 12 或 17, -23）。两个数字分别是 x 坐标和 y 坐标。
- 坐标空间 (Coordinate space)：显示对象中包含的坐标（其子元素所在的位置）的图形。
- 原点 (Origin)：坐标空间中的一个点， x 轴和 y 轴在此位置相交。该点的坐标为 0, 0。
- 点 (Point)：坐标空间中的一个位置。在 ActionScript 使用的二维坐标系中，点是按其 x 轴和 y 轴位置 （点坐标）来定义的。
- 注册点 (Registration point)：显示对象的坐标空间的原点（0, 0 坐标）。

- 缩放 (Scale)：相对于原始大小的对象大小。用作动词时，对象缩放是指伸展或缩小对象以更改其大小。
- 平移 (Translate)：将点的坐标从一个坐标空间更改为另一个坐标空间。
- 变形 (Transformation)：对图形的可视特性进行的调整，如旋转对象、改变其缩放比例、倾斜或扭曲其形状或者改变其颜色。
- X 轴 (X axis)：ActionScript 使用的二维坐标系中的水平轴。
- Y 轴 (Y axis)：ActionScript 使用的二维坐标系中的垂直轴。

## 完成本章中的示例

本章中的许多示例都说明了计算或更改值；这些示例中大部分都包括相应的 `trace()` 函数调用以说明代码的结果。要测试这些示例，请执行以下操作：

1. 创建一个空的 Flash 文档。
2. 在时间轴上选择一个关键帧。
3. 打开 "动作" 面板，将代码清单复制到 "脚本" 窗格中。
4. 使用 "控制" > "测试影片" 运行程序。

   您将在 "输出" 面板中看到该代码清单的 `trace` 函数的结果。

本章中某些示例说明了如何将变形应用于显示对象。对于这些示例而言，示例的结果可直观地看到而不是通过文本输出看到。要测试变形示例，请执行以下操作：

1. 创建一个空的 Flash 文档。
2. 在时间轴上选择一个关键帧。
3. 打开 "动作" 面板，将代码清单复制到 "脚本" 窗格中。
4. 在舞台上创建一个影片剪辑元件实例。例如，绘制一个形状，选中它，选择 "修改" > "转换为元件"，并给元件指定一个名称。
5. 在 "属性" 检查器中选中舞台影片剪辑，为实例指定一个实例名称。该名称应与用于显示示例代码清单中的对象的名称相匹配，例如，如果代码清单将变形应用于名为 `myDisplayObject` 的对象，则应将影片剪辑实例的名称也命名为 `myDisplayObject`。
6. 使用 "控制" > "测试影片" 运行程序。

   在屏幕上，您将看到将变形应用于代码清单中指定的对象后的结果。

测试示例代码清单的技术将在第 53 页的 "测试本章内的示例代码清单"中详细说明。

# 使用 Point 对象

Point 对象定义一对笛卡尔坐标。它表示二维坐标系中的某个位置。其中 *x* 表示水平轴，*y* 表示垂直轴。

要定义 Point 对象，请设置它的 x 和 y 属性，如下所示：

```
import flash.geom.*;
var pt1:Point = new Point(10, 20); // x == 10; y == 20
var pt2:Point = new Point();
pt2.x = 10;
pt2.y = 20;
```

## 确定两点之间的距离

可以使用 Point 类的 distance() 方法确定坐标空间两点之间的距离。例如，下面的代码确定同一显示对象容器中两个显示对象 （circle1 和 circle2）的注册点之间的距离：

```
import flash.geom.*;
var pt1:Point = new Point(circle1.x, circle1.y);
var pt2:Point = new Point(circle2.x, circle2.y);
var distance:Number = Point.distance(pt1, pt2);
```

## 平移坐标空间

如果两个显示对象位于不同的显示对象容器中，则它们可能位于不同的坐标空间。您可以使用 DisplayObject 类的 localToGlobal() 方法将坐标平移到舞台中相同（全局）坐标空间。例如，下面的代码确定不同显示对象容器中两个显示对象 （circle1 和 circle2）的注册点之间的距离：

```
import flash.geom.*;
var pt1:Point = new Point(circle1.x, circle1.y);
pt1 = circle1.localToGlobal(pt1);
var pt2:Point = new Point(circle1.x, circle1.y);
pt2 = circle2.localToGlobal(pt2);
var distance:Number = Point.distance(pt1, pt2);
```

同样，要确定名为 target 的显示对象的注册点与舞台上特定点之间的距离，您可以使用 DisplayObject 类的 localToGlobal() 方法：

```
import flash.geom.*;
var stageCenter:Point = new Point();
stageCenter.x = this.stage.stageWidth / 2;
stageCenter.y = this.stage.stageHeight / 2;
var targetCenter:Point = new Point(target.x, target.y);
targetCenter = target.localToGlobal(targetCenter);
var distance:Number = Point.distance(stageCenter, targetCenter);
```

## 按指定的角度和距离移动显示对象

您可以使用 Point 类的 `polar()` 方法将显示对象按特定角度移动特定距离。例如，下列代码按 60 度将 `myDisplayObject` 对象移动 100 个像素：

```
import flash.geom.*;
var distance:Number = 100;
var angle:Number = 2 * Math.PI * (90 / 360);
var translatePoint:Point = Point.polar(distance, angle);
myDisplayObject.x += translatePoint.x;
myDisplayObject.y += translatePoint.y;
```

## Point 类的其它用法

您可以将 Point 对象用于以下方法和属性：

| 类 | 方法或属性 | 说明 |
| --- | --- | --- |
| DisplayObjectContainer | `areInaccessibleObjectsUnderPoint()`<br>`getObjectsUnderPoint()` | 用于返回显示对象容器中某个点下的对象的列表。 |
| BitmapData | `hitTest()` | 用于定义 BitmapData 对象中的像素以及要检查点击的点。 |
| BitmapData | `applyFilter()`<br>`copyChannel()`<br>`merge()`<br>`paletteMap()`<br>`pixelDissolve()`<br>`threshold()` | 用于定义那些定义操作的矩形的位置。 |
| Matrix | `deltaTransformPoint()`<br>`transformPoint()` | 用于定义您要对其应用变形的点。 |
| Rectangle | `bottomRight`<br>`size`<br>`topLeft` | 用于定义这些属性。 |

# 使用 Rectangle 对象

Rectangle 对象定义一个矩形区域。 Rectangle 对象有一个位置，该位置由其左上角的 *x* 和 *y* 坐标以及 `width` 属性和 `height` 属性定义。通过调用 `Rectangle()` 构造函数可以定义新 Rectangle 对象的这些属性，如下所示：

```
import flash.geom.Rectangle;
var rx:Number = 0;
var ry:Number = 0;
var rwidth:Number = 100;
var rheight:Number = 50;
var rect1:Rectangle = new Rectangle(rx, ry, rwidth, rheight);
```

## 调整 Rectangle 对象的大小和进行重新定位

有多种方法调整 Rectangle 对象的大小和进行重新定位。

您可以通过更改 Rectangle 对象的 `x` 和 `y` 属性直接重新定位该对象。这对 Rectangle 对象的宽度或高度没有任何影响。

```
import flash.geom.Rectangle;
var x1:Number = 0;
var y1:Number = 0;
var width1:Number = 100;
var height1:Number = 50;
var rect1:Rectangle = new Rectangle(x1, y1, width1, height1);
trace(rect1) // (x=0, y=0, w=100, h=50)
rect1.x = 20;
rect1.y = 30;
trace(rect1); // (x=20, y=30, w=100, h=50)
```

如以下代码所示，如果更改 Rectangle 对象的 `left` 或 `top` 属性，也可以重新定位，并且该对象的 *x* 和 *y* 属性分别与 `left` 和 `top` 属性匹配。但是，Rectangle 对象的左下角位置不发生更改，所以调整了对象的大小。

```
import flash.geom.Rectangle;
var x1:Number = 0;
var y1:Number = 0;
var width1:Number = 100;
var height1:Number = 50;
var rect1:Rectangle = new Rectangle(x1, y1, width1, height1);
trace(rect1) // (x=0, y=0, w=100, h=50)
rect1.left = 20;
rect1.top = 30;
trace(rect1); // (x=30, y=20, w=70, h=30)
```

同样，如下面的示例所示，如果更改 Rectangle 对象的 bottom 或 right 属性，该对象的左上角位置不发生更改，所以相应地调整了对象的大小。

```
import flash.geom.Rectangle;
var x1:Number = 0;
var y1:Number = 0;
var width1:Number = 100;
var height1:Number = 50;
var rect1:Rectangle = new Rectangle(x1, y1, width1, height1);
trace(rect1) // (x=0, y=0, w=100, h=50)
rect1.right = 60;
trect1.bottom = 20;
trace(rect1); // (x=0, y=0, w=60, h=20)
```

也可以使用 offset() 方法重新定位 Rectangle 对象，如下所示：

```
import flash.geom.Rectangle;
var x1:Number = 0;
var y1:Number = 0;
var width1:Number = 100;
var height1:Number = 50;
var rect1:Rectangle = new Rectangle(x1, y1, width1, height1);
trace(rect1) // (x=0, y=0, w=100, h=50)
rect1.offset(20, 30);
trace(rect1); // (x=20, y=30, w=100, h=50)
```

offsetPt() 方法工作方式类似，只不过它是将 Point 对象作为参数，而不是将 $x$ 和 $y$ 偏移量值作为参数。

还可以使用 inflate() 方法调整 Rectangle 对象的大小，该方法包含两个参数，dx 和 dy。dx 参数表示矩形的左边和右边距中心的像素数，而 dy 参数表示矩形的顶边和底边距中心的像素数：

```
import flash.geom.Rectangle;
var x1:Number = 0;
var y1:Number = 0;
var width1:Number = 100;
var height1:Number = 50;
var rect1:Rectangle = new Rectangle(x1, y1, width1, height1);
trace(rect1) // (x=0, y=0, w=100, h=50)
rect1.inflate(6,4);
trace(rect1); // (x=-6, y=-4, w=112, h=58)
```

inflatePt() 方法作方式类似，只不过它是将 Point 对象作为参数，而不是将 dx 和 dy 的值作为参数。

# 确定 Rectangle 对象的联合和交集

可以使用 union() 方法来确定由两个矩形的边界形成的矩形区域：

```
import flash.display.*;
import flash.geom.Rectangle;
var rect1:Rectangle = new Rectangle(0, 0, 100, 100);
trace(rect1); // (x=0, y=0, w=100, h=100)
var rect2:Rectangle = new Rectangle(120, 60, 100, 100);
trace(rect2); // (x=120, y=60, w=100, h=100)
trace(rect1.union(rect2)); // (x=0, y=0, w=220, h=160)
```

可以使用 intersection() 方法来确定由两个矩形重叠区域形成的矩形区域：

```
import flash.display.*;
import flash.geom.Rectangle;
var rect1:Rectangle = new Rectangle(0, 0, 100, 100);
trace(rect1); // (x=0, y=0, w=100, h=100)
var rect2:Rectangle = new Rectangle(80, 60, 100, 100);
trace(rect2); // (x=120, y=60, w=100, h=100)
trace(rect1.intersection(rect2)); // (x=80, y=60, w=20, h=40)
```

使用 intersects() 方法查明两个矩形是否相交。也可以使用 intersects() 方法查明显示对象是否在舞台的某个区域中。例如，在下面的代码中，假定包含 circle 对象的显示对象容器的坐标空间与舞台的坐标空间相同。本示例说明如何使用 intersects() 方法来确定显示对象 circle 是否与由 target1 和 target2 Rectangle 对象定义的指定舞台区域相交：

```
import flash.display.*;
import flash.geom.Rectangle;
var circle:Shape = new Shape();
circle.graphics.lineStyle(2, 0xFF0000);
circle.graphics.drawCircle(250, 250, 100);
addChild(circle);
var circleBounds:Rectangle = circle.getBounds(stage);
var target1:Rectangle = new Rectangle(0, 0, 100, 100);
trace(circleBounds.intersects(target1)); // false
var target2:Rectangle = new Rectangle(0, 0, 300, 300);
trace(circleBounds.intersects(target2)); // true
```

同样，可以使用 intersects() 方法查明两个显示对象的边界矩形是否重叠。可以使用 DisplayObject 类的 getRect() 方法来包括显示对象笔触可添加到边界区域中的其它任何空间。

## Rectangle 对象的其它用法

Rectangle 对象可用于以下方法和属性:

| 类 | 方法或属性 | 描述 |
|---|---|---|
| BitmapData | `applyFilter()`、`colorTransform()`、`copyChannel()`、`copyPixels()`、`draw()`、`fillRect()`、`generateFilterRect()`、`getColorBoundsRect()`、`getPixels()`、`merge()`、`paletteMap()`、`pixelDissolve()`、`setPixels()` 和 `threshold()` | 用作某些参数的类型以定义 BitmapData 对象的区域。 |
| DisplayObject | `getBounds()`、`getRect()`、`scrollRect`、`scale9Grid` | 用作属性的数据类型或返回的数据类型。 |
| PrintJob | `addPage()` | 用于定义 `printArea` 参数。 |
| Sprite | `startDrag()` | 用于定义 `bounds` 参数。 |
| TextField | `getCharBoundaries()` | 用作返回值类型。 |
| Transform | `pixelBounds` | 用作数据类型。 |

# 使用 Matrix 对象

Matrix 类表示一个转换矩阵，它确定如何将点从一个坐标空间映射到另一个坐标空间。可以对显示对象执行不同的图形转换，方法是设置 Matrix 对象的属性，将该 Matrix 对象应用于 Transform 对象的 `matrix` 属性，然后应用该 Transform 对象作为显示对象的 `transform` 属性。这些转换函数包括平移（*x* 和 *y* 重新定位）、旋转、缩放和倾斜。

## 定义 Matrix 对象

虽然可以通过直接调整 Matrix 对象的属性（`a`、`b`、`c`、`d`、`tx` 和 `ty`）来定义矩阵，但更简单的方法是使用 `createBox()` 方法。使用此方法提供的参数可以直接定义生成的矩阵的缩放、旋转和平移效果。例如，下面的代码创建一个 Matrix 对象，具有效果是水平缩放 2.0、垂直缩放 3.0、旋转 45 度、向右移动（平移） 10 个像素并向下移动 20 个像素:

```
var matrix:Matrix = new Matrix();
var scaleX:Number = 2.0;
var scaleY:Number = 3.0;
var rotation:Number = 2 * Math.PI * (45 / 360);
var tx:Number = 10;
var ty:Number = 20;
matrix.createBox(scaleX, scaleY, rotation, tx, ty);
```

还可以使用 scale()、rotate() 和 translate() 方法调整 Matrix 对象的缩放、旋转和平移效果。请注意，这些方法合并了现有 Matrix 对象的值。例如，下面的代码调用两次 scale() 和 rotate() 方法以对 Matrix 对象进行设置，它将对象放大 4 倍并旋转 60 度：

```
var matrix:Matrix = new Matrix();
var rotation:Number = 2 * Math.PI * (30 / 360); // 30°
var scaleFactor:Number = 2;
matrix.scale(scaleFactor, scaleFactor);
matrix.rotate(rotation);
matrix.scale(scaleX, scaleY);
matrix.rotate(rotation);

myDisplayObject.transform.matrix = matrix;
```

要将倾斜转换应用到 Matrix 对象，请调整该对象的 b 或 c 属性。调整 b 属性将矩阵垂直倾斜，并调整 c 属性将矩阵水平倾斜。以下代码使用系数 2 垂直倾斜 myMatrix Matrix 对象：

```
var skewMatrix:Matrix = new Matrix();
skewMatrix.b = Math.tan(2);
myMatrix.concat(skewMatrix);
```

可以将矩阵转换应用到显示对象的 transform 属性。例如，以下代码将矩阵转换应用于名为 myDisplayObject 的显示对象：

```
var matrix:Matrix = myDisplayObject.transform.matrix;
var scaleFactor:Number = 2;
var rotation:Number = 2 * Math.PI * (60 / 360); // 60°
matrix.scale(scaleFactor, scaleFactor);
matrix.rotate(rotation);

myDisplayObject.transform.matrix = matrix;
```

第一行将 Matrix 对象设置为 myDisplayObject 显示对象所使用的现有转换矩阵（myDisplayObject 显示对象的 transformation 属性的 matrix 属性）。这样，调用的 Matrix 类方法将对显示对象的现有位置、缩放和旋转产生累积影响。



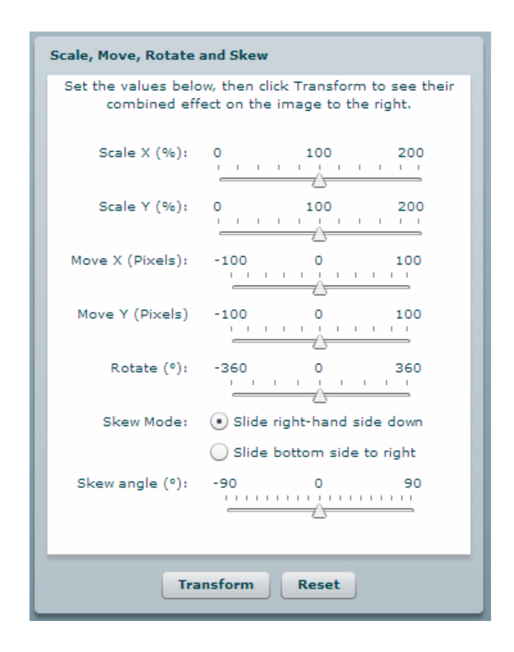
ColorTransform 类还包含在 flash.geometry 包中。该类用于设置 Transform 对象的 colorTransform 属性。由于它不应用任何种类的几何变形，本章中将不做讨论。有关详细信息，请参阅《ActionScript 3.0 语言和组件参考》中的 ColorTransform 类。

# 例如：将矩阵转换用于显示对象

DisplayObjectTransformer 范例应用程序说明许多使用 Matrix 类来变换显示对象的功能，包括：

- 旋转显示对象
- 缩放显示对象
- 平移（重新定位）显示对象
- 倾斜显示对象

该应用程序提供了接口，以调整矩阵转换的参数，如下所示：



当用户单击 “变形” 按钮时，应用程序将应用适当的变形。




*原始显示对象和旋转了 -45° 并缩放 50% 的显示对象*

要获取该范例的应用程序文件，请访问 www.adobe.com/go/learn_programmingAS3samples_flash_cn。可以在 Samples/DisplayObjectTransformer 中找到 DisplayObjectTransformer 应用程序文件。该应用程序包含以下文件：

| 文件 | 描述 |
|---|---|
| DisplayObjectTransformer.mxml<br>或<br>DisplayObjectTransformer.fla | Flash (FLA) 或 Flex (MXML) 中的主应用程序文件。 |
| com/example/programmingas3/geometry/MatrixTransformer.as | 一个类，包含用于应用矩阵转换的方法。 |
| img/ | 一个目录，包含应用程序使用的范例图像文件。 |

## 定义 MatrixTransformer 类

MatrixTransformer 类包含应用 Matrix 对象的几何变形的静态方法。

### transform() 方法

`transform()` 方法包含以下属性的参数：

- `sourceMatrix` — 此方法转换的输入矩阵
- `xScale` 和 `yScale` — $x$ 和 $y$ 缩放系数
- `dx` 和 `dy` — $x$ 和 $y$ 平移量 （以像素为单位）
- `rotation` — 旋转量 （以度为单位）

- skew — 倾斜系数（以百分比表示）
- skewType — 倾斜的方向，"right" 或 "left"

返回值为生成的矩阵。

transform() 方法调用下列类的静态方法：

- skew()
- scale()
- translate()
- rotate()

每种方法都返回应用了转换的源矩阵。

## skew() 方法

skew() 方法通过调整矩阵的 b 和 c 属性来倾斜矩阵。可选参数 unit 确定用于定义倾斜角度的单位，如果必要，该方法会将 angle 值转换为弧度：

```
if (unit == "degrees")
{
  angle = Math.PI * 2 * angle / 360;
}
if (unit == "gradients")
{
  angle = Math.PI * 2 * angle / 100;
}
```

创建并调整 skewMatrix Matrix 对象以应用倾斜转换。最初，它是恒等矩阵，如下所示：

```
var skewMatrix:Matrix = new Matrix();
```

skewSide 参数确定倾斜应用到的边。如果该参数设置为 "right"，则以下代码设置矩阵的 b 属性：

```
skewMatrix.b = Math.tan(angle);
```

否则，将通过调整矩阵的 c 属性来倾斜底边，如下所示：

```
skewMatrix.c = Math.tan(angle);
```

然后，通过将两个矩阵连接起来以将所产生的倾斜应用到现有矩阵，如下面的示例所示：

```
sourceMatrix.concat(skewMatrix);
return sourceMatrix;
```

## scale() 方法

如下面的示例所示，如果提供的缩放系数为百分比，则 `scale()` 方法将首先调整缩放系数，然后使用矩阵对象的 `scale()` 方法：

```
if (percent)
{
  xScale = xScale / 100;
  yScale = yScale / 100;
}
sourceMatrix.scale(xScale, yScale);
return sourceMatrix;
```

## translate() 方法

`translate()` 方法只需通过调用矩阵对象的 `translate()` 方法即可应用 `dx` 和 `dy` 平移系数，如下所示：

```
sourceMatrix.translate(dx, dy);
返回 sourceMatrix;
```

## rotate() 方法

`rotate()` 方法将输入的旋转系数转换为弧度（如果提供的是角度或渐变），然后调用矩阵对象的 `rotate()` 方法：

```
if (unit == "degrees")
{
  angle = Math.PI * 2 * angle / 360;
}
if (unit == "gradients")
{
  angle = Math.PI * 2 * angle / 100;
}
sourceMatrix.rotate(angle);
return sourceMatrix;
```

# 从应用程序中调用 MatrixTransformer.transform() 方法

应用程序提供了一个用户界面，以便从用户获得转换参数。然后将这些参数与显示对象的 `transform` 属性的 `matrix` 属性一起传递给 `Matrix.transform()` 方法，如下所示：

```
tempMatrix = MatrixTransformer.transform(tempMatrix,
                                         xScaleSlider.value,
                                         yScaleSlider.value,
                                         dxSlider.value,
                                         dySlider.value,
                                         rotationSlider.value,
                                         skewSlider.value,
                                         skewSide );
```

然后，该应用程序将返回值应用到显示对象的 transform 属性的 matrix 属性，从而触发转换：

```
img.content.transform.matrix = tempMatrix;
```

第 14 章

# 使用绘图 API

14

虽然导入的图像和插图非常重要，但您可以使用一项称为绘图 API 的功能 （用于在 ActionScript 中绘制线条和形状）随时启动计算机中的应用程序，这就相当于一个空白画布，您可以在上面创建所需的任何图像。能够创建自己的图形可为您的应用程序提供广阔的前景。通过使用本章中介绍的方法，您可以创建绘图程序、制作交互的动画效果，或以编程方式创建您自己的用户界面元素，等等。

## 目录

# 绘图 API 使用基础知识

## 使用绘图 API 简介

绘图 API 是 ActionScript 中的一项内置功能的名称，您可以使用该功能来创建矢量图形（直线、曲线、形状、填充和渐变），并使用 ActionScript 在屏幕上显示它们。flash.display.Graphics 类提供了这一功能。您可以在任何 Shape、 Sprite 或 MovieClip 实例中使用 ActionScript 进行绘制（使用其中的每个类中定义的 `graphics` 属性）。（实际上，每个类的 `graphics` 属性都是 Graphics 类的实例。）

如果刚刚开始学习使用代码进行绘制，可以使用 Graphics 类中包含的几种方法来简化绘制常见形状（如圆、椭圆、矩形以及带圆角的矩形）的过程。您可以将它们作为空线条或填充形状进行绘制。当您需要更高级的功能时，还可以使用 Graphics 类中包含的用于绘制直线和二次贝塞尔曲线的方法，您可以将这些方法与 Math 类中的三角函数配合使用来创建所需的任何形状。

## 常见绘图 API 任务

以下是您可能需要在 ActionScript 中使用绘图 API 完成的任务，本章对这些任务进行了介绍：

- 定义线条样式和填充样式以绘制形状
- 绘制直线和曲线
- 使用方法来绘制形状（如圆、椭圆和矩形）
- 使用渐变线条和填充进行绘制
- 定义矩阵以创建渐变
- 将三角函数与绘图 API 配合使用
- 将绘图 API 与动画相结合

## 重要概念和术语

以下参考列表包含将会在本章中遇到的重要术语：

- 锚点 (Anchor point)：二次贝塞尔曲线的两个端点之一。
- 控制点 (Control point)：该点定义了二次贝塞尔曲线的弯曲方向和弯曲量。弯曲的线绝不会到达控制点；但是，曲线就好像朝着控制点方向进行绘制的。
- 坐标空间 (Coordinate space)：显示对象中包含的坐标（其子元素所在的位置）的图形。
- 填充 (Fill)：形状内的实心部分，它包含一条用颜色填充的线条，或者整个形状都没有轮廓。

- 渐变 (Gradient)：此颜色是指从一种颜色逐渐过渡到一种或多种其它颜色（相对于纯色而言）。
- 点 (Point)：坐标空间中的一个位置。在 ActionScript 使用的二维坐标系中，点是按其 x 轴和 y 轴位置（点坐标）来定义的。
- 二次贝塞尔曲线 (Quadratic Bézier curve)：一种由特定数学公式定义的曲线类型。在这种类型的曲线中，曲线形状是根据锚点（曲线端点）和控制点（定义曲线的弯曲方向和弯曲量）的位置计算的。
- 缩放 (Scale)：相对于原始大小的对象大小。用作动词时，对象缩放是指伸展或缩小对象以更改其大小。
- 笔触 (Stroke)：形状的轮廓部分，它包含一条用颜色填充的线条，或未填充的形状的多个线条。
- 平移 (Translate)：将点的坐标从一个坐标空间更改为另一个坐标空间。
- X 轴 (X axis)：ActionScript 使用的二维坐标系中的水平轴。
- Y 轴 (Y axis)：ActionScript 使用的二维坐标系中的垂直轴。

## 完成本章中的示例

学习本章的过程中，您可能想要自己动手测试一些示例代码清单。由于本章涉及绘制可视内容，因此测试代码清单包括运行代码以及在创建的 SWF 中查看结果。要测试代码清单，请执行以下操作：

1. 创建一个空的 Flash 文档。
2. 在时间轴上选择一个关键帧。
3. 打开 “动作” 面板，将代码清单复制到 “脚本” 窗格中。
4. 使用 “控制” > “测试影片” 运行程序。

   您将在所创建的 SWF 文件中看到代码清单的结果。

# 了解 Graphics 类

每个 Shape、Sprite 和 MovieClip 对象都具有一个 `graphics` 属性，它是 Graphics 类的一个实例。Graphics 类包含用于绘制线条、填充和形状的属性和方法。如果要将显示对象仅用作内容绘制画布，则可以使用 Shape 实例。 Shape 实例的性能优于其它用于绘制的显示对象，因为它不会产生 Sprite 和 MovieClip 类中的附加功能的开销。如果希望能够在显示对象上绘制图形内容，并且还希望该对象包含其它显示对象，则可以使用 Sprite 实例。有关确定用于各种任务的显示对象的详细信息，请参阅第 338 页的“选择 DisplayObject 子类”。

# 绘制直线和曲线

使用 Graphics 实例进行的所有绘制均基于包含直线和曲线的基本绘制。因此，必须使用一系列相同的步骤来执行所有 ActionScript 绘制：

- 定义线条和填充样式
- 设置初始绘制位置
- 绘制直线、曲线和形状 （可选择移动绘制点）
- 如有必要，完成创建填充

## 定义线条和填充样式

要使用 Shape、 Sprite 或 MovieClip 实例的 `graphics` 属性进行绘制，您必须先定义在绘制时使用的样式（线条大小和颜色、填充颜色）。就像使用 Adobe Flash CS3 Professional 或其它绘图应用程序中的绘制工具一样，使用 ActionScript 进行绘制时，可以使用笔触进行绘制，也可以不使用笔触；可以使用填充颜色进行绘制，也可以不使用填充颜色。您可以使用 `lineStyle()` 或 `lineGradientStyle()` 方法来指定笔触的外观。要创建纯色线条，请使用 `lineStyle()` 方法。调用此方法时，您指定的最常用的值是前三个参数：线条粗细、颜色以及 Alpha。例如，该行代码指示名为 `myShape` 的 Shape 对象绘制 2 个像素粗、红色 (0x990000) 以及 75% 不透明的线条：

```
myShape.graphics.lineStyle(2, 0x990000, .75);
```

Alpha 参数的默认值为 1.0 (100%)，因此，如果需要完全不透明的线条，可以将该参数的值保持不变。`lineStyle()` 方法还接受两个用于像素提示和缩放模式的额外参数；有关使用这些参数的详细信息，请参阅 《ActionScript 3.0 语言和组件参考》中的 `Graphics.lineStyle()` 方法的描述。

要创建渐变线条，请使用 `lineGradientStyle()` 方法。第 393 页的 “创建渐变线条和填充”中介绍了此方法。

如果要创建填充形状，请在开始绘制之前调用 `beginFill()`、`beginGradientFill()` 或 `beginBitmapFill()` 方法。其中的最基本方法 `beginFill()` 接受以下两个参数：填充颜色以及填充颜色的 Alpha 值（可选）。例如，如果要绘制具有纯绿色填充的形状，应使用以下代码 （假设在名为 `myShape` 的对象上进行绘制）：

```
myShape.graphics.beginFill(0x00FF00);
```

调用任何填充方法时，将隐式地结束任何以前的填充，然后再开始新的填充。调用任何指定笔触样式的方法时，将替换以前的笔触，但不会改变以前指定的填充，反之亦然。

指定了线条样式和填充属性后，下一步是指示绘制的起始点。 Graphics 实例具有一个绘制点，就像在一张纸上的钢笔尖一样。无论绘制点位于什么位置，它都是开始执行下一个绘制动作的位置。最初， Graphics 对象将它绘制时所在对象的坐标空间中的点 (0, 0) 作为起始绘制点。要在其它点开始进行绘制，您可以先调用 `moveTo()` 方法，然后再调用绘制方法之一。这类似于将钢笔尖从纸上抬起，然后将其移到新位置。

确定绘制点后，可通过使用对绘制方法 `lineTo()`（用于绘制直线）和 `curveTo()`（用于绘制曲线）的一系列调用来进行绘制。

> **提示** 在进行绘制时，可随时调用 `moveTo()` 方法，将绘制点移到新位置而不进行绘制。

在进行绘制时，如果已指定了填充颜色，可以指示 Adobe Flash Player 调用 `endFill()` 方法来结束填充。如果绘制的形状没有闭合 （换句话说，在调用 `endFill()` 时，绘制点不在形状的起始点），调用 `endFill()` 方法时， Flash Player 将自动绘制一条直线以闭合形状，该直线从当前绘制点到最近一次 `moveTo()` 调用中指定的位置。如果已开始填充并且没有调用 `endFill()`，调用 `beginFill()`（或其它填充方法之一）时，将关闭当前填充并开始新的填充。

## 绘制直线

调用 `lineTo()` 方法时，Graphics 对象将绘制一条直线，该直线从当前绘制点到指定为方法调用中的两个参数的坐标，以便使用指定的线条样式进行绘制。例如，该行代码将绘制点放在点 (100, 100) 上，然后绘制一条到点 (200, 200) 的直线：

```
myShape.graphics.moveTo(100, 100);
myShape.graphics.lineTo(200, 200);
```

以下示例绘制红色和绿色三角形，其高度为 100 个像素：

```
var triangleHeight:uint = 100;
var triangle:Shape = new Shape();

// red triangle, starting at point 0, 0
triangle.graphics.beginFill(0xFF0000);
triangle.graphics.moveTo(triangleHeight/2, 0);
triangle.graphics.lineTo(triangleHeight, triangleHeight);
triangle.graphics.lineTo(0, triangleHeight);
triangle.graphics.lineTo(triangleHeight/2, 0);

// green triangle, starting at point 200, 0
triangle.graphics.beginFill(0x00FF00);
triangle.graphics.moveTo(200 + triangleHeight/2, 0);
triangle.graphics.lineTo(200 + triangleHeight, triangleHeight);
triangle.graphics.lineTo(200, triangleHeight);
triangle.graphics.lineTo(200 + triangleHeight/2, 0);

this.addChild(triangle);
```

## 绘制曲线

curveTo() 方法可以绘制二次贝塞尔曲线。这将绘制一个连接两个点（称为锚点）的弧，同时向第三个点（称为控制点）弯曲。Graphics 对象使用当前绘制位置作为第一个锚点。调用 curveTo() 方法时，将传递以下四个参数：控制点的 x 和 y 坐标，后跟第二个锚点的 x 和 y 坐标。例如，以下代码绘制一条曲线，它从点 (100, 100) 开始，到点 (200, 200) 结束。由于控制点位于点 (175, 125)，因此，这会创建一条曲线，它先向右移动，然后向下移动：

```
myShape.graphics.moveTo(100, 100);
myShape.graphics.curveTo(175, 125, 200, 200);
```

以下示例绘制红色和绿色圆形对象，其宽度和高度均为 100 个像素。请注意，由于二次贝塞尔方程式所具有的特性，这些对象并不是完美的圆：

```
var size:uint = 100;
var roundObject:Shape = new Shape();

// red circular shape
roundObject.graphics.beginFill(0xFF0000);
roundObject.graphics.moveTo(size / 2, 0);
roundObject.graphics.curveTo(size, 0, size, size / 2);
roundObject.graphics.curveTo(size, size, size / 2, size);
roundObject.graphics.curveTo(0, size, 0, size / 2);
roundObject.graphics.curveTo(0, 0, size / 2, 0);

// green circular shape
roundObject.graphics.beginFill(0x00FF00);
roundObject.graphics.moveTo(200 + size / 2, 0);
roundObject.graphics.curveTo(200 + size, 0, 200 + size, size / 2);
roundObject.graphics.curveTo(200 + size, size, 200 + size / 2, size);
roundObject.graphics.curveTo(200, size, 200, size / 2);
roundObject.graphics.curveTo(200, 0, 200 + size / 2, 0);

this.addChild(roundObject);
```

# 使用内置方法绘制形状

为了便于绘制常见形状（如圆、椭圆、矩形以及带圆角的矩形），ActionScript 3.0 中提供了用于绘制这些常见形状的方法。它们是 Graphics 类的 drawCircle()、drawEllipse()、drawRect()、drawRoundRect() 和 drawRoundRectComplex() 方法。这些方法可用于替代 lineTo() 和 curveTo() 方法。但要注意，在调用这些方法之前，您仍需指定线条和填充样式。

以下示例重新创建绘制红色、绿色以及蓝色正方形的示例，其宽度和高度均为 100 个像素。以下代码使用 drawRect() 方法，并且还指定了填充颜色的 Alpha 为 50% (0.5)：

```
var squareSize:uint = 100;
var square:Shape = new Shape();
square.graphics.beginFill(0xFF0000, 0.5);
```

```
square.graphics.drawRect(0, 0, squareSize, squareSize);
square.graphics.beginFill(0x00FF00, 0.5);
square.graphics.drawRect(200, 0, squareSize, squareSize);
square.graphics.beginFill(0x0000FF, 0.5);
square.graphics.drawRect(400, 0, squareSize, squareSize);
square.graphics.endFill();
this.addChild(square);
```

在 Sprite 或 MovieClip 对象中，使用 `graphics` 属性创建的绘制内容始终出现在该对象包含的所有子级显示对象的后面。另外，`graphics` 属性内容不是单独的显示对象，因此，它不会出现在 Sprite 或 MovieClip 对象的子级列表中。例如，以下 Sprite 对象使用其 `graphics` 属性来绘制圆，并且其子级显示对象列表中包含一个 TextField 对象：

```
var mySprite:Sprite = new Sprite();
mySprite.graphics.beginFill(0xFFCC00);
mySprite.graphics.drawCircle(30, 30, 30);
var label:TextField = new TextField();
label.width = 200;
label.text = "They call me mellow yellow...";
label.x = 20;
label.y = 20;
mySprite.addChild(label);
this.addChild(mySprite);
```

请注意， TextField 将出现在使用 graphics 对象绘制的圆的上面。

# 创建渐变线条和填充

graphics 对象也可以绘制渐变笔触和填充，而不是纯色笔触和填充。渐变笔触是使用 `lineGradientStyle()` 方法创建的；渐变填充是使用 `beginGradientFill()` 方法创建的。

这两种方法接受相同的参数。前四个参数是必需的，即类型、颜色、Alpha 以及比率。其余四个参数是可选的，但对于高级自定义非常有用。

- 第一个参数指定要创建的渐变类型。可接受的值是 `GradientFill.LINEAR` 或 `GradientFill.RADIAL`。
- 第二个参数指定要使用的颜色值的数组。在线性渐变中，将从左向右排列颜色。在放射状渐变中，将从内到外排列颜色。数组颜色的顺序表示在渐变中绘制颜色的顺序。
- 第三个参数指定前一个参数中相应颜色的 Alpha 透明度值。
- 第四个参数指定比率或每种颜色在渐变中的重要程度。可接受的值范围是 0-255。这些值并不表示任何宽度或高度，而是表示在渐变中的位置； 0 表示渐变开始， 255 表示渐变结束。比率数组必须按顺序增加，并且包含的条目数与第二个和第三个参数中指定的颜色和 Alpha 数组相同。

虽然第五个参数（转换矩阵）是可选的，但通常会使用该参数，因为它提供了一种简便且有效的方法来控制渐变外观。此参数接受 Matrix 实例。为渐变创建 Matrix 对象的最简单方法是使用 Matrix 类的 `createGradientBox()` 方法。

## 定义 Matrix 对象以用于渐变

可以使用 flash.display.Graphics 类的 `beginGradientFill()` 和 `lineGradientStyle()` 方法来定义在形状中使用的渐变。定义渐变时，需要提供一个矩阵作为这些方法的其中一个参数。

定义矩阵的最简单方法是使用 Matrix 类的 `createGradientBox()` 方法，该方法创建一个用于定义渐变的矩阵。可以使用传递给 `createGradientBox()` 方法的参数来定义渐变的缩放、旋转和位置。`createGradientBox()` 方法接受以下参数：

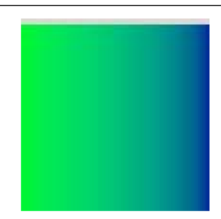
- 渐变框宽度：渐变扩展到的宽度（以像素为单位）
- 渐变框高度：渐变扩展到的高度（以像素为单位）
- 渐变框旋转：将应用于渐变的旋转角度（以弧度为单位）
- 水平平移：将渐变水平移动的距离（以像素为单位）
- 垂直平移：将渐变垂直移动的距离（以像素为单位）

例如，假设渐变具有以下特性：

- `GradientType.LINEAR`
- 绿色和蓝色这两种颜色（`ratios` 数组设置为 `[0, 255]`）
- `SpreadMethod.PAD`
- `InterpolationMethod.LINEAR_RGB`

下面的示例显示的是几种渐变，如图所示，它们的 `createGradientBox()` 方法的 `rotation` 参数不同，但所有其它设置是相同的：

```
width = 100;
height = 100;
rotation = 0;
tx = 0;
ty = 0;
```



```
width = 100;
height = 100;
rotation = Math.PI/4; // 45°
tx = 0;
ty = 0;
```

```
width = 100;
height = 100;
rotation = Math.PI/2; // 90°
tx = 0;
ty = 0;
```

下面的示例显示的是绿到蓝线性渐变的效果，如图所示，它们的 `createGradientBox()` 方法的 `rotation`、`tx` 和 `ty` 参数不同，但所有其它设置是相同的：

```
width = 50;
height = 100;
rotation = 0;
tx = 0;
ty = 0;
```

```
width = 50;
height = 100;
rotation = 0
tx = 50;
ty = 0;
```

```
width = 100;
height = 50;
rotation = Math.PI/2; // 90°
tx = 0;
ty = 0;
```

```
width = 100;
height = 50;
rotation = Math.PI/2; // 90°
tx = 0;
ty = 50;
```

createGradientBox() 方法的 width、height、tx 和 ty 参数也会影响“放射状”渐变填充的大小和位置，如下面的示例所示：

```
width = 50;
height = 100;
rotation = 0;
tx = 25;
ty = 0;
```

下面的代码生成了所示的最后一个放射状渐变：

```
import flash.display.Shape;
import flash.display.GradientType;
import flash.geom.Matrix;

var type:String = GradientType.RADIAL;
var colors:Array = [0x00FF00, 0x000088];
var alphas:Array = [1, 1];
var ratios:Array = [0, 255];
var spreadMethod:String = SpreadMethod.PAD;
var interp:String = InterpolationMethod.LINEAR_RGB;
var focalPtRatio:Number = 0;

var matrix:Matrix = new Matrix();
var boxWidth:Number = 50;
var boxHeight:Number = 100;
var boxRotation:Number = Math.PI/2; // 90°
var tx:Number = 25;
var ty:Number = 0;
matrix.createGradientBox(boxWidth, boxHeight, boxRotation, tx, ty);
```

```
var square:Shape = new Shape;
square.graphics.beginGradientFill(type,
                colors,
                alphas,
                ratios,
                matrix,
                spreadMethod,
                interp,
                focalPtRatio);
square.graphics.drawRect(0, 0, 100, 100);
addChild(square);
```

请注意，渐变填充的宽度和高度是由渐变矩阵的宽度和高度决定的，而不是由使用 Graphics 对象绘制的宽度和高度决定的。使用 Graphics 对象进行绘制时，您绘制的内容位于渐变矩阵中的这些坐标处。即使使用 Graphics 对象的形状方法之一（如 `drawRect()`），渐变也不会将其自身伸展到绘制的形状的大小；必须在渐变矩阵本身中指定渐变的大小。

下面说明了渐变矩阵的尺寸和绘图本身的尺寸之间的视觉差异：

```
var myShape:Shape = new Shape();
var gradientBoxMatrix:Matrix = new Matrix();
gradientBoxMatrix.createGradientBox(100, 40, 0, 0, 0);
myShape.graphics.beginGradientFill(GradientType.LINEAR, [0xFF0000, 0x00FF00,
  0x0000FF], [1, 1, 1], [0, 128, 255], gradientBoxMatrix);
myShape.graphics.drawRect(0, 0, 50, 40);
myShape.graphics.drawRect(0, 50, 100, 40);
myShape.graphics.drawRect(0, 100, 150, 40);
myShape.graphics.endFill();
this.addChild(myShape);
```

该代码绘制三个具有相同填充样式（使用平均分布的红色、绿色和蓝色指定的）的渐变。这些渐变是使用 `drawRect()` 方法绘制的，像素宽度分别为 50、100 和 150。`beginGradientFill()` 方法中指定的渐变矩阵是使用像素宽度 100 创建的。这意味着，第一个渐变仅包含渐变色谱的一半，第二个渐变包含全部色谱，而第三个渐变包含全部色谱以及向右扩展的额外 50 蓝色像素。

`lineGradientStyle()` 方法的工作方式与 `beginGradientFill()` 类似，所不同的是，除了定义渐变外，您还必须在绘制之前使用 `lineStyle()` 方法指定笔触粗细。以下代码绘制一个带有红色、绿色和蓝色渐变笔触的框：

```
var myShape:Shape = new Shape();
var gradientBoxMatrix:Matrix = new Matrix();
gradientBoxMatrix.createGradientBox(200, 40, 0, 0, 0);
myShape.graphics.lineStyle(5, 0);
myShape.graphics.lineGradientStyle(GradientType.LINEAR, [0xFF0000, 0x00FF00,
  0x0000FF], [1, 1, 1], [0, 128, 255], gradientBoxMatrix);
myShape.graphics.drawRect(0, 0, 200, 40);
this.addChild(myShape);
```

有关 Matrix 类的详细信息，请参阅第 379 页的 “使用 Matrix 对象”。

# 将 Math 类与绘制方法配合使用

Graphics 对象可以绘制圆和正方形，但也可以绘制更复杂的形状，尤其是在将绘制方法与 Math 类的属性和方法配合使用时。 Math 类包含人们通常很感兴趣的数学常量，如 `Math.PI` （约等于 3.14159265...），此常量表示圆的周长与其直径的比率。它还包含三角函数的方法，其中包括 `Math.sin()`、`Math.cos()` 和 `Math.tan()` 等。使用这些方法和常量绘制形状可产生更动态的视觉效果，尤其是用于重复或递归时。

Math 类的很多方法都要求以弧度为单位来测量圆弧，而不是使用角度。 Math 类的一个常见用途是在这两种类型的单位之间进行转换：

```
var degrees = 121;
var radians = degrees * Math.PI / 180;
trace(radians) // 2.111848394913139
```

以下示例创建一个正弦波和余弦波以重点说明给定值的 `Math.sin()` 和 `Math.cos()` 方法之间的差异。

```
var sinWavePosition = 100;
var cosWavePosition = 200;
var sinWaveColor:uint = 0xFF0000;
var cosWaveColor:uint = 0x00FF00;
var waveMultiplier:Number = 10;
var waveStretcher:Number = 5;

var i:uint;
for(i = 1; i < stage.stageWidth; i++)
{
  var sinPosY:Number = Math.sin(i / waveStretcher) * waveMultiplier;
  var cosPosY:Number = Math.cos(i / waveStretcher) * waveMultiplier;

  graphics.beginFill(sinWaveColor);
  graphics.drawRect(i, sinWavePosition + sinPosY, 2, 2);
  graphics.beginFill(cosWaveColor);
  graphics.drawRect(i, cosWavePosition + cosPosY, 2, 2);
}
```

# 使用绘图 API 进行动画处理

使用绘图 API 创建内容的一个优点是，您并不限于将内容放置一次。可通过保留和修改用于绘制的变量来修改所绘制的内容。您可以通过更改变量和重绘（在一段帧上或使用计时器）来利用原有的动画。

例如，以下代码更改每个经过的帧（通过侦听 Event.ENTER_FRAME 事件）的显示内容以增加当前度数，指示 graphics 对象清除内容并在更新位置进行重绘。

```
stage.frameRate = 31;

var currentDegrees:Number = 0;
var radius:Number = 40;
var satelliteRadius:Number = 6;

var container:Sprite = new Sprite();
container.x = stage.stageWidth / 2;
container.y = stage.stageHeight / 2;
addChild(container);
var satellite:Shape = new Shape();
container.addChild(satellite);

addEventListener(Event.ENTER_FRAME, doEveryFrame);

function doEveryFrame(event:Event):void
{
  currentDegrees += 4;
  var radians:Number = getRadians(currentDegrees);
  var posX:Number = Math.sin(radians) * radius;
  var posY:Number = Math.cos(radians) * radius;
  satellite.graphics.clear();
  satellite.graphics.beginFill(0);
  satellite.graphics.drawCircle(posX, posY, satelliteRadius);
}
function getRadians(degrees:Number):Number
{
    return degrees * Math.PI / 180;
}
```

要产生明显不同的效果，您可以尝试修改代码开头的初始种子变量（currentDegrees、radius 和 satelliteRadius）。例如，尝试缩小 radius 变量和 / 或增大 totalSatellites 变量。这只是一个说明绘图 API 如何创建可视显示内容（其复杂性掩盖了创建简便性）的示例。

# 示例：Algorithmic Visual Generator

Algorithmic Visual Generator 示例在舞台上动态绘制几个 “卫星”，即在圆形轨道中运动的圆形物。要阐述的功能包括：

- 使用绘图 API 绘制具有动态外观的基本形状
- 将用户交互与绘图中使用的属性相关联
- 清除每个帧中的舞台内容并重绘以利用原有的动画

上一小节中的示例使用 `Event.ENTER_FRAME` 事件对唯一的 “卫星”进行动画处理。该示例在此基础之上进一步扩展，生成一个包含一系列滑块的控制面板，这些滑块会立即更新若干卫星的可视显示内容。此示例将代码格式设置为外部类，并将卫星创建代码包装在循环中，以将对每个卫星的引用存储在 `satellites` 数组中。

要获取该范例的应用程序文件，请访问
www.adobe.com/go/learn_programmingAS3samples_flash_cn。
可以在 Samples/AlgorithmicVisualGenerator 文件夹中找到应用程序文件。此文件夹包含以下文件：

| 文件 | 说明 |
| --- | --- |
| AlgorithmicVisualGenerator.fla | Flash 中的主应用程序文件 (FLA)。 |
| com/example/programmingas3/algorithmic/AlgorithmicVisualGenerator.as | 此类提供应用程序的主要功能，其中包括在舞台上绘制卫星，以及从控制面板中响应事件以更新影响卫星绘制的变量。 |
| com/example/programmingas3/algorithmic/ControlPanel.as | 此类管理用户与几个滑块之间的交互并在发生此类交互时调度事件。 |
| com/example/programmingas3/algorithmic/Satellite.as | 此类表示在轨道中围绕中心点旋转的显示对象，并包含与其当前绘制状态有关的属性。 |

## 设置侦听器

应用程序先创建三个侦听器。第一个侦听器侦听从控制面板中调度的事件：必须对卫星进行重新构建。第二个侦听器侦听对 SWF 文件的舞台大小的更改。第三个侦听器侦听 SWF 文件中每个经过的帧，并使用 `doEveryFrame()` 函数进行重绘。

## 创建卫星

在设置这些侦听器后，将调用 build() 函数。此函数先调用 clear() 函数，后者清空 satellites 数组，并清除以前在舞台上绘制的任何内容。这是必要的，因为每当控制面板发送事件来执行此操作时，都可能会重新调用 build() 函数，例如在颜色设置已发生变化时。在这种情况下，必须删除并重新创建卫星。

该函数随后创建一些卫星，并设置创建所需的初始属性，如 position 变量（从轨道中的随机位置开始）和 color 变量（在本示例中，该变量在创建卫星后不会发生改变）。

创建每个卫星时，会将对它的引用添加到 satellites 数组中。调用 doEveryFrame() 函数时，它将更新此数组中的所有卫星。

## 更新卫星位置

doEveryFrame() 函数是应用程序动画过程的核心。将为每个帧调用该函数，频率等于 SWF 文件的帧频。由于绘制变量略微发生了变化，因此，这会利用原有的动画外观。

该函数先清除以前绘制的所有内容，然后再重绘背景。接下来，它循环访问每个卫星容器，增加每个卫星的 position 属性以及更新 radius 和 orbitRadius 属性（这些属性可能由于用户与控制面板的交互而发生了变化）。最后，通过调用 Satellite 类的 draw() 方法，在新位置对卫星进行更新。

请注意，计数器 i 最多只增加到 visibleSatellites 变量。这是因为，如果用户通过控制面板限制了显示的卫星数，则不会重绘循环中的其余卫星，而应将其隐藏起来。这种情况发生在紧靠负责绘制的循环后面的循环中。

当 doEveryFrame() 函数完成时，将在屏幕上的位置中更新 visibleSatellites 数量。

## 响应用户交互

用户交互是通过控制面板发生的，它是由 ControlPanel 类管理的。此类设置一个侦听器，并为每个滑块设置单独的最小、最大和默认值。当用户移动这些滑块时，将调用 changeSetting() 函数。此函数更新控制面板的属性。如果更改需要重新构建显示内容，则会调度一个事件，随后将在主应用程序文件中对该事件进行处理。当控制面板设置发生变化时，doEveryFrame() 函数将使用更新的变量绘制每个卫星。

## 进一步自定义

本示例只是概要介绍了使用绘图 API 生成可视内容方面的基础知识。它使用相对较少的几行代码来创建似乎非常复杂的交互式体验。尽管如此，您还是可以对本示例进行较小的改动以进行扩展。下面列出了一些很好的方法：

- `doEveryFrame()` 函数可以增加卫星的颜色值。
- `doEveryFrame()` 函数可能会随着时间的推移缩小或增大卫星半径。
- 卫星并不一定是圆形的，例如，可以使用 Math 类根据正弦波来移动其半径。
- 卫星可以使用碰撞检测来检查是否与其它卫星重叠。

可以将绘图 API 作为在 Flash 创作环境中创建视觉效果的替代方法，以便在运行时绘制基本形状。但是，它也可以用来创建涵盖范围很广且无法手动创建的各种视觉效果。通过使用绘图 API 和一些数学函数， ActionScript 作者可能实现很多意想不到的创作效果。

第 15 章

# 过滤显示对象

15

以往，对位图图像应用滤镜效果是专门图像编辑软件（如 Adobe Photoshop® 和 Adobe Fireworks®）的范畴。ActionScript 3.0 包括 flash.filters 包，它包含一系列位图效果滤镜类，允许开发人员以编程方式对位图应用滤镜并显示对象，以达到图形处理应用程序中所具有的许多相同效果。

目录



## 过滤显示对象的基础知识

### 过滤显示对象简介

为应用程序添加优美效果的一种方式是添加简单的图形效果，如在图片后面添加投影可产生三维视觉效果，在按钮周围添加发光可表示该按钮当前处于活动状态。ActionScript 3.0 包括九种可应用于任何显示对象或 BitmapData 实例的滤镜。滤镜的范围从基本滤镜（如投影和发光滤镜）到用于创建各种效果的复杂滤镜（如置换图滤镜和卷积滤镜）。

### 常见过滤任务

以下是您在 ActionScript 中使用滤镜时将要完成的任务：

- 创建滤镜
- 对显示对象应用滤镜
- 对 BitmapData 实例中的图像数据应用滤镜
- 删除对象中的滤镜

- 创造各种滤镜效果，如：
  - 发光
  - 模糊
  - 投影
  - 锐化
  - 置换
  - 边缘检测
  - 浮雕
  - 和其它效果

## 重要概念和术语

以下参考列表包含您将会在本章中遇到的重要术语：

- 斜角：通过使两侧的像素变亮并使相对两侧的像素变暗所形成的一个边缘，它可以产生常用于凸起或凹进按钮和类似图形的三维边界效果。
- 卷积：通过使用各种比率将每个像素的值与其周围某些像素或全部像素的值合并以使图像中的像素发生扭曲。
- 置换：将图像中的像素移动到新位置。
- 矩阵: 用于执行特定数学计算的网格数字，使用方法是对网格中的数字应用不同的值，然后合并结果。

## 完成本章中的示例

学习本章的过程中，您可能想要测试所提供的示例代码清单。由于本章涉及创建和操作可视内容，因此测试代码包括运行代码以及在创建的 SWF 中查看结果。几乎所有示例都或者使用绘图 API 创建内容，或者加载要应用滤镜的图像。

要测试本章中的代码，请执行以下操作：

1. 创建一个空的 Flash 文档。
2. 在时间轴上选择一个关键帧。
3. 打开 “动作” 面板，将代码复制到 “脚本” 窗格中。
4. 使用 “控制” > “测试影片” 运行程序。

   您将在所创建的 SWF 文件中看到代码的结果。

几乎所有示例代码都包括用于创建位图图像的代码，因此您可以直接测试代码，而无需提供任何位图内容。或者，您也可以更改代码清单，以加载自己的图像并使用这些图像替代示例中的图像。

# 创建和应用滤镜

使用滤镜可以对位图和显示对象应用从投影到斜角和模糊等各种效果。由于将每个滤镜定义为一个类，因此应用滤镜涉及创建滤镜对象的实例，这与构造任何其它对象并没有区别。创建了滤镜对象的实例后，通过使用该对象的 `filters` 属性可以很容易地将此实例应用于显示对象；如果是 BitmapData 对象，可以使用 `applyFilter()` 方法。

## 创建新滤镜

若要创建新滤镜对象，只需调用所选的滤镜类的构造函数方法即可。例如，若要创建新的 DropShadowFilter 对象，请使用以下代码：

```
import flash.filters.DropShadowFilter;
var myFilter:DropShadowFilter = new DropShadowFilter();
```

虽然此处没有显示参数，但 `DropShadowFilter()` 构造函数（与所有滤镜类的构造函数一样）接受多个可用于自定义滤镜效果外观的可选参数。

## 应用滤镜

构造滤镜对象后，可以将其应用于显示对象或 BitmapData 对象；应用滤镜的方式取决于您应用该滤镜的对象。

### 对显示对象应用滤镜

对显示对象应用滤镜效果时，可以通过 `filters` 属性应用这些效果。显示对象的 `filters` 属性是一个 Array 实例，其中的元素是应用于该显示对象的滤镜对象。若要对显示对象应用单个滤镜，请创建该滤镜实例，将其添加到 Array 实例，再将该 Array 对象分配给显示对象的 `filters` 属性：

```
import flash.display.Bitmap;
import flash.display.BitmapData;
import flash.filters.DropShadowFilter;

// 创建 bitmapData 对象并将它呈现在屏幕上
var myBitmapData:BitmapData = new BitmapData(100,100,false,0xFFFF3300);
var myDisplayObject:Bitmap = new Bitmap(myBitmapData);
addChild(myDisplayObject);

// 创建 DropShadowFilter 实例。
var dropShadow:DropShadowFilter = new DropShadowFilter();

// 创建滤镜数组，通过将滤镜作为参数传递给 Array() 构造函数，
// 将该滤镜添加到数组中。
var filtersArray:Array = new Array(dropShadow);
```

```
// 将滤镜数组分配给显示对象以便应用滤镜。
myDisplayObject.filters = filtersArray;
```

如果要为该对象分配多个滤镜，只需在将 Array 实例分配给 filters 属性之前将所有滤镜添加到该实例即可。可以通过将多个对象作为参数传递给 Array 的构造函数，将多个对象添加到 Array。例如，以下代码对上面创建的显示对象应用斜角滤镜和发光滤镜：

```
import flash.filters.BevelFilter;
import flash.filters.GlowFilter;

// 创建滤镜并将其添加到数组。
var bevel:BevelFilter = new BevelFilter();
var glow:GlowFilter = new GlowFilter();
var filtersArray:Array = new Array(bevel, glow);

// 将滤镜数组分配给显示对象以便应用滤镜。
myDisplayObject.filters = filtersArray;
```

**注意**

在创建包含滤镜的数组时，您可以使用 new Array() 构造函数创建该数组 （如前面示例所示），也可以使用 Array 文本语法将滤镜括在方括号 ([]) 中。例如，下面这行代码：

```
var filters:Array = new Array(dropShadow, blur);
```

与下面这行代码效果相同：

```
var filters:Array = [dropShadow, blur];
```

如果对显示对象应用多个滤镜，则会按顺序以累积方式应用这些滤镜。例如，如果滤镜数组有两个元素：先添加的斜角滤镜和后添加的投影滤镜，则投影滤镜既会应用于斜角滤镜，也会应用于显示对象。这是由于投影滤镜在滤镜数组中处于第二的位置。如果想要以非累积方式应用滤镜，则必须对显示对象的新副本应用每个滤镜。

如果只为显示对象分配一个或几个滤镜，则可以先创建该滤镜实例，然后在单个语句中将该实例分配给该对象。例如，下面的一行代码将模糊滤镜应用于名为 myDisplayObject 的显示对象：

```
myDisplayObject.filters = [new BlurFilter()];
```

上面的代码使用 Array 文本语法（方括号）创建一个 Array 实例，创建一个新的 BlurFilter 实例作为 Array 中的一个元素，并将该 Array 分配给名为 myDisplayObject 的显示对象的 filters 属性。

## 删除显示对象中的滤镜

删除显示对象中的所有滤镜非常简单，只需为 filters 属性分配一个 null 值即可：

```
myDisplayObject.filters = null;
```

如果您已对一个对象应用了多个滤镜，并且只想要删除其中一个滤镜，则必须完成多个步骤才能更改 filters 属性数组。有关详细信息，请参阅第 407 页的 “在运行时更改滤镜”。

## 对 BitmapData 对象应用滤镜

对 BitmapData 对象应用滤镜需要使用 BitmapData 对象的 `applyFilter()` 方法：

```
myBitmapData.applyFilter(sourceBitmapData);
```

`applyFilter()` 方法会对源 BitmapData 对象应用滤镜，从而生成一个新的、应用滤镜的图像。此方法不会修改原始的源图像；而是将对源图像应用滤镜的结果存储在调用 `applyFilter()` 方法的 BitmapData 实例中。

# 滤镜的工作原理

显示对象过滤是通过将原始对象的副本缓存为透明位图来工作的。

将滤镜应用于显示对象后，只要此对象具有有效的滤镜列表，Adobe Flash Player 就会将该对象缓存为位图。然后，将此位图用作所有后续应用的滤镜效果的原始图像。

每个显示对象通常包含两个位图：一个包含原始未过滤的源显示对象，另一个用于过滤后的最终图像。呈现时使用最终图像。只要显示对象不发生更改，最终图像就不需要更新。

# 使用滤镜的潜在问题

使用滤镜时要记住几个导致混淆或问题的潜在根源。在下面几节中介绍了这些问题。

## 滤镜和位图缓存

若要对显示对象应用滤镜，必须启用该对象的位图缓存。在对 `cacheAsBitmap` 属性设置为 `false` 的显示对象应用滤镜时，Flash Player 会自动将该对象的 `cacheAsBitmap` 属性的值设置为 `true`。如果您以后删除了该显示对象中的所有滤镜，Flash Player 会将 `cacheAsBitmap` 属性重置为最后设置的值。

## 在运行时更改滤镜

如果已经对显示对象应用了一个或多个滤镜，则无法向 `filters` 属性数组添加其它滤镜。若要添加或更改应用的这组滤镜，需要创建整个滤镜数组的副本，然后对此（临时）数组进行修改。然后，将此数组重新分配给显示对象的 `filters` 属性，这样才能将滤镜应用于该对象。以下代码演示了此过程。开始时，对名为 `myDisplayObject` 的显示对象应用了发光滤镜，之后，在单击该显示对象时，调用 `addFilters()` 函数。在此函数中，将两个其它滤镜应用于 myDisplayObject：

```
import flash.events.MouseEvent;
import flash.filters.*;

myDisplayObject.filters = [new GlowFilter()];

function addFilters(event:MouseEvent):void
```

```
{
    // 制作滤镜数组的副本。
    var filtersCopy:Array = myDisplayObject.filters;

    // 对滤镜进行需要的更改 (在本例中为添加滤镜)。
    filtersCopy.push(new BlurFilter());
    filtersCopy.push(new DropShadowFilter());

    // 通过将数组重新分配给 filters 属性来应用更改。
    myDisplayObject.filters = filtersCopy;
}

myDisplayObject.addEventListener(MouseEvent.CLICK, addFilters);
```

## 滤镜和对象变形

在显示对象的边框矩形之外的任何过滤区域（例如投影）都不能视为可进行点击检测（确定实例是否与其它实例重叠或交叉）的表面。由于 DisplayObject 类的点击检测方法是基于矢量的，因此无法在位图结果上执行点击检测。例如，如果您对按钮实例应用斜角滤镜，则在该实例的斜角部分，点击检测不可用。

滤镜不支持缩放、旋转和倾斜；如果过滤的显示对象本身进行了缩放（如果 `scaleX` 和 `scaleY` 不是 100%），则滤镜效果将不随该实例缩放。这意味着，实例的原始形状将旋转、缩放或倾斜；而滤镜不随实例一起旋转、缩放或倾斜。

可以使用滤镜给实例添加动画，以形成理想的效果，或者嵌套实例并使用 BitmapData 类使滤镜动起来，以获得此效果。

## 滤镜和位图对象

对 BitmapData 对象应用滤镜时，`cacheAsBitmap` 属性会自动设置为 `true`。通过这种方式，滤镜实际上是应用于对象的副本而不是原始对象。

之后，会将此副本放在主显示（原始对象）上，尽量接近最近的像素。如果原始位图的边框发生更改，则会从头重新创建过滤的副本位图，而不进行伸展或扭曲。

如果清除了显示对象的所有滤镜，`cacheAsBitmap` 属性会重置为应用滤镜之前的值。

# 可用的显示滤镜

ActionScript 3.0 包括 9 个可用于显示对象和 BitmapData 对象的滤镜类：

- 斜角滤镜（BevelFilter 类）
- 模糊滤镜（BlurFilter 类）
- 投影滤镜（DropShadowFilter 类）
- 发光滤镜（GlowFilter 类）
- 渐变斜角滤镜（GradientBevelFilter 类）
- 渐变发光滤镜（GradientGlowFilter 类）
- 颜色矩阵滤镜（ColorMatrixFilter 类）
- 卷积滤镜（ConvolutionFilter 类）
- 置换图滤镜（DisplacementMapFilter 类）

前六个滤镜是简单滤镜，可用于创建一种特定效果，并可以对效果进行某种程度的自定义。可以使用 ActionScript 应用这六个滤镜，也可以在 Adobe Flash CS3 Professional 中使用"滤镜"面板将其应用于对象。因此，即使您要使用 ActionScript 应用滤镜，如果有 Flash 创作工具，也可以使用可视界面快速尝试不同的滤镜和设置，弄清楚如何创建需要的效果。

最后三个滤镜仅在 ActionScript 中可用。这些滤镜（颜色矩阵滤镜、卷积滤镜和置换图滤镜）所创建的效果具有极其灵活的形式；它们不仅仅可以进行优化以生成单一的效果，而且还具有强大的功能和灵活性。例如，如果为卷积滤镜的矩阵选择不同的值，则它可用于创建模糊、浮雕、锐化、查找颜色边缘、变形等效果。

不管是简单滤镜还是复杂滤镜，每个滤镜都可以使用其属性进行自定义。通常，您有两种方法用于设置滤镜属性。所有滤镜都允许通过向滤镜对象的构造函数传递参数值来设置属性。或者，不管您是否通过传递参数来设置滤镜属性，都可以在以后通过设置滤镜对象的属性值来调整滤镜。多数示例代码清单都直接设置属性，以便更易于按照示例进行操作。不过，您通常可以通过在滤镜对象的构造函数中以参数的形式传递值来获得同样的结果。有关每个滤镜的特性、其属性和构造函数的更多详细信息，请参阅《ActionScript 3.0 语言和组件参考》中的 flash.filters 包的列表。

## 斜角滤镜

BevelFilter 类允许您对过滤的对象添加三维斜面边缘。此滤镜可使对象的硬角或边缘具有硬角或边缘被凿削或呈斜面的效果。

BevelFilter 类属性允许您自定义斜角的外观。您可以设置加亮和阴影颜色、斜角边缘模糊、斜角角度和斜角边缘的位置，甚至可以创建挖空效果。

以下示例加载外部图像并对它应用斜角滤镜。

```
import flash.display.*;
import flash.filters.BevelFilter;
import flash.filters.BitmapFilterQuality;
import flash.filters.BitmapFilterType;
import flash.net.URLRequest;

// 将图像加载到舞台上。
var imageLoader:Loader = new Loader();
var url:String = "http://www.helpexamples.com/flash/images/image3.jpg";
var urlReq:URLRequest = new URLRequest(url);
imageLoader.load(urlReq);
addChild(imageLoader);

// 创建斜角滤镜并设置滤镜属性。
var bevel:BevelFilter = new BevelFilter();

bevel.distance = 5;
bevel.angle = 45;
bevel.highlightColor = 0xFFFF00;
bevel.highlightAlpha = 0.8;
bevel.shadowColor = 0x666666;
bevel.shadowAlpha = 0.8;
bevel.blurX = 5;
bevel.blurY = 5;
bevel.strength = 5;
bevel.quality = BitmapFilterQuality.HIGH;
bevel.type = BitmapFilterType.INNER;
bevel.knockout = false;

// 对图像应用滤镜。
imageLoader.filters = [bevel];
```

## 模糊滤镜

BlurFilter 类可使显示对象及其内容具有涂抹或模糊的效果。模糊效果可以用于产生对象不在焦点之内的视觉效果，也可以用于模拟快速运动，比如运动模糊。通过将模糊滤镜的 `quality` 属性设置为低，可以模拟轻轻离开焦点的镜头效果。将 `quality` 属性设置为高会产生类似高斯模糊的平滑模糊效果。

以下示例使用 Graphics 类的 `drawCircle()` 方法创建一个圆形对象并对它应用模糊滤镜：

```
import flash.display.Sprite;
import flash.filters.BitmapFilterQuality;
import flash.filters.BlurFilter;

// 绘制一个圆。
var redDotCutout:Sprite = new Sprite();
redDotCutout.graphics.lineStyle();
```

```
redDotCutout.graphics.beginFill(0xFF0000);
redDotCutout.graphics.drawCircle(145, 90, 25);
redDotCutout.graphics.endFill();

// 将该圆添加到显示列表中。
addChild(redDotCutout);

// 对矩形应用模糊滤镜。
var blur:BlurFilter = new BlurFilter();
blur.blurX = 10;
blur.blurY = 10;
blur.quality = BitmapFilterQuality.MEDIUM;
redDotCutout.filters = [blur];
```

## 投影滤镜

投影给人一种目标对象上方有独立光源的印象。可以修改此光源的位置和强度，以产生各种不同的投影效果。

投影滤镜使用与模糊滤镜的算法相似的算法。主要区别是投影滤镜有更多的属性，您可以修改这些属性来模拟不同的光源属性 （如 Alpha、颜色、偏移和亮度）。

投影滤镜还允许您对投影的样式应用自定义变形选项，包括内侧或外侧阴影和挖空（也称为剪切块）模式。

以下代码创建方框 sprite 并对它应用投影滤镜：

```
import flash.display.Sprite;
import flash.filters.DropShadowFilter;

// 绘制一个框。
var boxShadow:Sprite = new Sprite();
boxShadow.graphics.lineStyle(1);
boxShadow.graphics.beginFill(0xFF3300);
boxShadow.graphics.drawRect(0, 0, 100, 100);
boxShadow.graphics.endFill();
addChild(boxShadow);

// 对该框应用投影滤镜。
var shadow:DropShadowFilter = new DropShadowFilter();
shadow.distance = 10;
shadow.angle = 25;

// 您也可以设置其它属性，如投影颜色、
// Alpha、模糊量、强度、品质和
// 内侧阴影和挖空效果选项。

boxShadow.filters = [shadow];
```

## 发光滤镜

GlowFilter 类对显示对象应用加亮效果，使显示对象看起来像是被下方的灯光照亮，可创造出一种柔和发光效果。

与投影滤镜类似，发光滤镜包括的属性可修改光源的距离、角度和颜色，以产生各种不同效果。 GlowFilter 还有多个选项用于修改发光样式，包括内侧或外侧发光和挖空模式。

以下代码使用 Sprite 类创建一个交叉对象并对它应用发光滤镜：

```
import flash.display.Sprite;
import flash.filters.BitmapFilterQuality;
import flash.filters.GlowFilter;

// 创建一个交叉图形。
var crossGraphic:Sprite = new Sprite();
crossGraphic.graphics.lineStyle();
crossGraphic.graphics.beginFill(0xCCCC00);
crossGraphic.graphics.drawRect(60, 90, 100, 20);
crossGraphic.graphics.drawRect(100, 50, 20, 100);
crossGraphic.graphics.endFill();
addChild(crossGraphic);

// 对该交叉形状应用发光滤镜。
var glow:GlowFilter = new GlowFilter();
glow.color = 0x009922;
glow.alpha = 1;
glow.blurX = 25;
glow.blurY = 25;
glow.quality = BitmapFilterQuality.MEDIUM;

crossGraphic.filters = [glow];
```

## 渐变斜角滤镜

GradientBevelFilter 类允许您对显示对象或 BitmapData 对象应用增强的斜角效果。在斜角上使用渐变颜色可以大大改善斜角的空间深度，使边缘产生一种更逼真的三维外观效果。

以下代码使用 Shape 类的 `drawRect()` 方法创建一个矩形对象，并对它应用渐变斜角滤镜。

```
import flash.display.Shape;
import flash.filters.BitmapFilterQuality;
import flash.filters.GradientBevelFilter;

// 绘制一个矩形。
var box:Shape = new Shape();
box.graphics.lineStyle();
box.graphics.beginFill(0xFEFE78);
box.graphics.drawRect(100, 50, 90, 200);
box.graphics.endFill();
```

```
// 对该矩形应用渐变斜角。
var gradientBevel:GradientBevelFilter = new GradientBevelFilter();

gradientBevel.distance = 8;
gradientBevel.angle = 225; // 反向 45 度
gradientBevel.colors = [0xFFFFCC, 0xFEFE78, 0x8F8E01];
gradientBevel.alphas = [1, 0, 1];
gradientBevel.ratios = [0, 128, 255];
gradientBevel.blurX = 8;
gradientBevel.blurY = 8;
gradientBevel.quality = BitmapFilterQuality.HIGH;

// 其它属性允许您设置滤镜强度和设置用于
// 内侧斜角和挖空效果的选项。

box.filters = [gradientBevel];

// 将该图形添加到显示列表中。
addChild(box);
```

## 渐变发光滤镜

GradientGlowFilter 类允许您对显示对象或 BitmapData 对象应用增强的发光效果。该效果可使您更好地控制发光颜色，因而可产生一种更逼真的发光效果。另外，渐变发光滤镜还允许您对对象的内侧、外侧或上侧边缘应用渐变发光。

以下示例在舞台上绘制一个圆形，并对它应用渐变发光滤镜。当您进一步向右和向下移动鼠标时，会分别增加水平和垂直方向的模糊量。此外，只要您在舞台上单击，就会增加模糊的强度。

```
import flash.events.MouseEvent;
import flash.filters.BitmapFilterQuality;
import flash.filters.BitmapFilterType;
import flash.filters.GradientGlowFilter;

// 创建一个新的 Shape 实例。
var shape:Shape = new Shape();

// 绘制形状。
shape.graphics.beginFill(0xFF0000, 100);
shape.graphics.moveTo(0, 0);
shape.graphics.lineTo(100, 0);
shape.graphics.lineTo(100, 100);
shape.graphics.lineTo(0, 100);
shape.graphics.lineTo(0, 0);
shape.graphics.endFill();

// 在舞台上定位该形状。
```

```
addChild(shape);
shape.x = 100;
shape.y = 100;

// 定义渐变发光。
var gradientGlow:GradientGlowFilter = new GradientGlowFilter();
gradientGlow.distance = 0;
gradientGlow.angle = 45;
gradientGlow.colors = [0x000000, 0xFF0000];
gradientGlow.alphas = [0, 1];
gradientGlow.ratios = [0, 255];
gradientGlow.blurX = 10;
gradientGlow.blurY = 10;
gradientGlow.strength = 2;
gradientGlow.quality = BitmapFilterQuality.HIGH;
gradientGlow.type = BitmapFilterType.OUTER;

// 定义侦听两个事件的函数。
function onClick(event:MouseEvent):void
{
  gradientGlow.strength++;
  shape.filters = [gradientGlow];
}

function onMouseMove(event:MouseEvent):void
{
  gradientGlow.blurX = (stage.mouseX / stage.stageWidth) * 255;
  gradientGlow.blurY = (stage.mouseY / stage.stageHeight) * 255;
  shape.filters = [gradientGlow];
}
stage.addEventListener(MouseEvent.CLICK, onClick);
stage.addEventListener(MouseEvent.MOUSE_MOVE, onMouseMove);
```

## 示例：合并基本滤镜

以下代码示例使用与 Timer 合并的多个基本滤镜创建重复动作，以形成一种动画交通信号灯模拟效果。

```
import flash.display.Shape;
import flash.events.TimerEvent;
import flash.filters.BitmapFilterQuality;
import flash.filters.BitmapFilterType;
import flash.filters.DropShadowFilter;
import flash.filters.GlowFilter;
import flash.filters.GradientBevelFilter;
import flash.utils.Timer;

var count:Number = 1;
var distance:Number = 8;
```

```
var angleInDegrees:Number = 225; // 反向 45 度
var colors:Array = [0xFFFFCC, 0xFEFE78, 0x8F8E01];
var alphas:Array = [1, 0, 1];
var ratios:Array = [0, 128, 255];
var blurX:Number = 8;
var blurY:Number = 8;
var strength:Number = 1;
var quality:Number = BitmapFilterQuality.HIGH;
var type:String = BitmapFilterType.INNER;
var knockout:Boolean = false;

// 绘制交通信号灯的矩形背景。
var box:Shape = new Shape();
box.graphics.lineStyle();
box.graphics.beginFill(0xFEFE78);
box.graphics.drawRect(100, 50, 90, 200);
box.graphics.endFill();

// 绘制 3 个圆形表示三个交通灯。
var stopLight:Shape = new Shape();
stopLight.graphics.lineStyle();
stopLight.graphics.beginFill(0xFF0000);
stopLight.graphics.drawCircle(145,90,25);
stopLight.graphics.endFill();

var cautionLight:Shape = new Shape();
cautionLight.graphics.lineStyle();
cautionLight.graphics.beginFill(0xFF9900);
cautionLight.graphics.drawCircle(145,150,25);
cautionLight.graphics.endFill();

var goLight:Shape = new Shape();
goLight.graphics.lineStyle();
goLight.graphics.beginFill(0x00CC00);
goLight.graphics.drawCircle(145,210,25);
goLight.graphics.endFill();

// 将这些图形添加到显示列表中。
addChild(box);
addChild(stopLight);
addChild(cautionLight);
addChild(goLight);

// 对交通信号灯矩形应用渐变斜角。
var gradientBevel:GradientBevelFilter = new GradientBevelFilter(distance,
  angleInDegrees, colors, alphas, ratios, blurX, blurY, strength, quality,
  type, knockout);
box.filters = [gradientBevel];

// 创建内侧阴影（用于灯关闭时）和发光
```

```
// （用于灯打开时）。
var innerShadow:DropShadowFilter = new DropShadowFilter(5, 45, 0, 0.5, 3, 3,
  1, 1, true, false);
var redGlow:GlowFilter = new GlowFilter(0xFF0000, 1, 30, 30, 1, 1, false,
  false);
var yellowGlow:GlowFilter = new GlowFilter(0xFF9900, 1, 30, 30, 1, 1, false,
  false);
var greenGlow:GlowFilter = new GlowFilter(0x00CC00, 1, 30, 30, 1, 1, false,
  false);

// 设置灯的起始状态（绿灯打开，红灯 / 黄灯关闭）。
stopLight.filters = [innerShadow];
cautionLight.filters = [innerShadow];
goLight.filters = [greenGlow];

// 根据计数值交换滤镜。
function trafficControl(event:TimerEvent):void
{
  if (count == 4)
  {
    count = 1;
  }

  switch (count)
  {
    case 1:
      stopLight.filters = [innerShadow];
      cautionLight.filters = [yellowGlow];
      goLight.filters = [innerShadow];
      break;
    case 2:
      stopLight.filters = [redGlow];
      cautionLight.filters = [innerShadow];
      goLight.filters = [innerShadow];
      break;
    case 3:
      stopLight.filters = [innerShadow];
      cautionLight.filters = [innerShadow];
      goLight.filters = [greenGlow];
      break;
  }

  count++;
}

// 创建一个计时器，每隔 3 秒钟交换一次滤镜。
var timer:Timer = new Timer(3000, 9);
timer.addEventListener(TimerEvent.TIMER, trafficControl);
timer.start();
```

## 颜色矩阵滤镜

ColorMatrixFilter 类用于操作过滤对象的颜色和 Alpha 值。它允许您进行饱和度更改、色相旋转（将调色板从一个颜色范围移动到另一个颜色范围）、将亮度更改为 Alpha，以及生成其它颜色操作效果，方法是使用一个颜色通道中的值，并将这些值潜移默化地应用于其它通道。

从概念上来说，滤镜将逐一处理源图像中的像素，并将每个像素分为红、绿、蓝和 Alpha 组件。然后，用每个值乘以颜色矩阵中提供的值，将结果加在一起以确定该像素将显示在屏幕上的最终颜色值。滤镜的 `matrix` 属性是一个由 20 个数字组成的数组，用于计算最终颜色。有关用于计算颜色值的特定算法的详细信息，请参阅 《ActionScript 3.0 语言和组件参考》中说明 ColorMatrixFilter 类的 `matrix` 属性的条目。

有关颜色矩阵滤镜的其它信息和示例，请参阅 Adobe 开发人员中心网站上提供的 "Using Matrices for Transformations, Color Adjustments, and Convolution Effects in Flash"（在 Flash 中使用矩阵实现变形、颜色调整和卷积效果）文章。

## 卷积滤镜

ConvolutionFilter 类可用于对 BitmapData 对象或显示对象应用广泛的图像变形，如模糊、边缘检测、锐化、浮雕和斜角。

从概念上来说，卷积滤镜会逐一处理源图像中的每个像素，并使用像素和它周围的像素的值来确定该像素的最终颜色。指定为数值数组的矩阵可以指示每个特定邻近像素的值对最终结果值具有何种程度的影响。

最常用的矩阵类型是 3 x 3 矩阵。此矩阵包括九个值：

```
N N N
N P N
N N N
```

Flash Player 对特定像素应用卷积滤镜时，它会考虑像素本身的颜色值（本示例中的"P"），以及周围像素的值 （本示例中的 "N"）。而通过设置矩阵中的值，可以指定特定像素在影响生成的图像方面所具有的优先级。

例如，使用卷积滤镜时应用的以下矩阵，会保持图像原样：

```
0 0 0
0 1 0
0 0 0
```

图像保持不变的原因是，在决定最终像素颜色时，原始像素的值相对强度为 1，而周围像素的值相对强度为 0 （意味着它们的颜色不影响最终图像）。

同样，下面的这个矩阵会使图像的像素向左移动一个像素：

```
0 0 0
0 0 1
0 0 0
```

请注意，在本例中，像素本身不影响最终图像上显示在该位置的像素的最终值，而只使用右侧的像素值来确定像素的结果值。

在 ActionScript 中，您可以通过组合一个包含值的 Array 实例和两个指定矩阵中行数和列数的属性来创建矩阵。以下示例加载一个图像，完成加载该图像后，使用前面列表中的矩阵对该图像应用卷积滤镜：

```
// 将图像加载到舞台上。
var loader:Loader = new Loader();
var url:URLRequest = new URLRequest("http://www.helpexamples.com/flash/
  images/image1.jpg");
loader.load(url);
this.addChild(loader);

function applyFilter(event:MouseEvent):void
{
  // 创建卷积矩阵。
  var matrix:Array = [ 0, 0, 0,
                       0, 0, 1,
                       0, 0, 0 ];

  var convolution:ConvolutionFilter = new ConvolutionFilter();
  convolution.matrixX = 3;
  convolution.matrixY = 3;
  convolution.matrix = matrix;
  convolution.divisor = 1;

  loader.filters = [convolution];
}

loader.addEventListener(MouseEvent.CLICK, applyFilter);
```

此代码中没有明确显示使用除矩阵中 1 或 0 以外的值将会产生的效果。例如，同一矩阵中如果右侧位置的数字是 8 而不是 1，则它会执行同样的动作 （向左移动像素）。此外，它还影响图像的颜色，使颜色加亮了 8 倍。这是因为最终像素颜色值的计算方法是用原始像素颜色乘以矩阵值，将这些值加在一起，再除以滤镜的 `divisor` 属性的值。请注意，在示例代码中，`divisor` 属性设置为 1。通常，如果想让颜色的明亮度与原始图像保持基本相同，应让 divisor 等于矩阵值之和。因此，如果矩阵的值相加为 8，并且除数为 1，结果图像将比原始图像明亮约 8 倍。

虽然此矩阵的效果不是很明显，但可以使用其它矩阵值产生各种不同效果。以下是几组标准矩阵值集合，用于使用 3 x 3 矩阵产生不同效果：

■ 基本模糊（除数 5）：

```
0 1 0
1 1 1
0 1 0
```

■ 锐化（除数 1)：

```
 0, -1, 0
-1, 5, -1
 0, -1, 0
```

■ 边缘检测（除数 1）：

```
 0, -1, 0
-1, 4, -1
 0, -1, 0
```

■ 浮雕效果（除数 1）：

```
-2, -1, 0
-1, 1, 1
 0, 1, 2
```

请注意，这些效果中多数的除数均为 1。这是因为负矩阵值加上正矩阵值的最后结果为 1（在边缘检测中，除数为 0，但 `divisor` 属性的值不能为 0）。

# 置换图滤镜

DisplacementMapFilter 类使用 BitmapData 对象（称为置换图图像）中的像素值在新对象上执行置换效果。通常，置换图图像与将要应用滤镜的实际显示对象或 BitmapData 实例不同。置换效果包括置换过滤的图像中的像素，也就是说，将这些像素移开原始位置一定距离。此滤镜可用于产生移位、扭曲或斑点效果。

应用于给定像素的置换位置和置换量由置换图图像的颜色值确定。使用滤镜时，除了指定置换图图像外，还要指定以下值，以便控制置换图图像中计算置换的方式：

■ 映射点：过滤图像上的位置，在该点将应用置换滤镜的左上角。如果只想对图像的一部分应用滤镜，可以使用此值。

■ X 组件：影响像素的 x 位置的置换图图像的颜色通道。

■ Y 组件：影响像素的 y 位置的置换图图像的颜色通道。

■ X 缩放比例：指定 x 轴置换强度的乘数值。

■ Y 缩放比例：指定 y 轴置换强度的乘数值。

■ 滤镜模式：确定在移开像素后形成的空白区域中，Flash Player 应执行什么操作。在 DisplacementMapFilterMode 类中定义为常量的选项可以显示原始像素（滤镜模式 IGNORE）、从图像的另一侧环绕像素（滤镜模式 WRAP，这是默认设置）、使用最近的移位像素（滤镜模式 CLAMP）或用颜色填充空间（滤镜模式 COLOR）。

若要对置换图滤镜的工作原理有一个基本了解，请查看下面的基本示例。在以下代码中，将加载一个图像，完成加载后使图像在舞台上居中并对它应用置换图滤镜，使整个图像中的像素向左水平移位。

```
import flash.display.BitmapData;
import flash.display.Loader;
import flash.events.MouseEvent;
import flash.filters.DisplacementMapFilter;
import flash.geom.Point;
import flash.net.URLRequest;

// 将图像加载到舞台上。
var loader:Loader = new Loader();
var url:URLRequest = new URLRequest("http://www.helpexamples.com/flash/
  images/image3.jpg");
loader.load(url);
this.addChild(loader);

var mapImage:BitmapData;
var displacementMap:DisplacementMapFilter;

// This function is called when the image finishes loading.
function setupStage(event:Event):void
{
  // 在舞台上将加载的图像居中。
  loader.x = (stage.stageWidth - loader.width) / 2;
  loader.y = (stage.stageHeight - loader.height) / 2;

  // 创建置换图图像。
  mapImage = new BitmapData(loader.width, loader.height, false, 0xFF0000);

  // 创建置换滤镜。
  displacementMap = new DisplacementMapFilter();
  displacementMap.mapBitmap = mapImage;
  displacementMap.mapPoint = new Point(0, 0);
  displacementMap.componentX = BitmapDataChannel.RED;
  displacementMap.scaleX = 250;
  loader.filters = [displacementMap];
}

loader.contentLoaderInfo.addEventListener(Event.COMPLETE, setupStage);
```

用于定义置换的属性有：

- 置换图位图：置换位图是由代码创建的新的 BitmapData 实例。它的尺寸与加载的图像的尺寸匹配 （因此会将置换应用于整个图像）。用纯红色像素填充此实例。
- 映射点：将此值设置为点 0, 0，使置换再次应用于整个图像。
- X 组件：此值设置为常量 `BitmapDataChannel.RED`，表示置换图位图的红色值将决定沿着 x 轴置换像素的程度 （像素的移动程度）。
- X 缩放比例：此值设置为 250。由于全部置换量 （与全红的置换图图像的距离）仅使图像置换很小的量（大约为一个像素的一半），因此，如果将此值设置为 1，图像只会水平移动 0.5 个像素。将此值设置为 250，图像将移动大约 125 个像素。

这些设置可使过滤图像的像素向左移动 250 个像素。移动的方向 （向左或向右）和移动量取决于置换图图像中的像素的颜色值。从概念上来说， Flash Player 会逐一处理过滤图像的像素 （至少处理将应用滤镜的区域中的像素，在本例中指所有像素），并对每个像素执行以下操作：

1. Flash Player 在置换图图像中查找相应的像素。例如，当 Flash Player 计算过滤图像左上角像素的置换量时，会在置换图图像左上角中查找像素。
2. Flash Player 确定置换图像素中指定颜色通道的值。在本例中，x 组件颜色通道是红色通道，因此 Flash Player 将查看置换图图像中该像素所在位置处的红色通道所对应的值。由于置换图图像是纯红色的，因此像素的红色通道为 0xFF 或 255。会将此值用作置换值。
3. Flash Player 比较置换值和 “中间”值 （127，它是 0 和 255 之间的中间值）。如果置换值低于中间值，则像素正向移位（x 置换向右； y 置换向下）。另一方面，如果置换值高于中间值 （如本示例），则像素负向移位 （x 置换向左； y 置换向上）。为更精确起见， Flash Player 会从 127 中减去置换值，结果 （正或负）即是应用的相对置换量。
4. 最后， Flash Player 通过确定相对置换值所表示的完全置换量的百分比来确定实际置换量。在本例中，全红色意味着 100% 置换。然后用 x 缩放比例值或 y 缩放比例值乘以该百分比，以确定将应用的置换像素数。在本示例中， 100% 乘以一个乘数 250 可以确定置换量 （大约为向左移动 125 个像素）。

由于没有为 y 组件和 y 缩放比例指定值，因此使用默认值（不发生置换），这就是图像在垂直方向不移位的原因。

由于在本示例中使用了默认滤镜模式设置 `WRAP`，因此在像素向左移位时，会用移到图像左边缘以外的像素填充右侧空白区域。您可以对此设置试用不同的值以查看不同的效果。例如，如果您在设置置换属性的代码部分中（在 `loader.filters = [displacementMap]` 行之前）添加以下一行内容，则会使图像在舞台上产生涂抹的效果：

```
displacementMap.mode = DisplacementMapFilterMode.CLAMP;
```

下面是一个更为复杂的示例，代码清单中使用置换图滤镜在图像上创建放大镜效果：

```
import flash.display.Bitmap;
import flash.display.BitmapData;
import flash.display.BitmapDataChannel;
import flash.display.GradientType;
import flash.display.Loader;
import flash.display.Shape;
import flash.events.MouseEvent;
import flash.filters.DisplacementMapFilter;
import flash.filters.DisplacementMapFilterMode;
import flash.geom.Matrix;
import flash.geom.Point;
import flash.net.URLRequest;

// 创建共同构成置换图图像的
// 渐变圆
var radius:uint = 50;

var type:String = GradientType.LINEAR;
var redColors:Array = [ 0xFF0000, 0x000000 ];
var blueColors:Array = [ 0x0000FF, 0x000000 ];
var alphas:Array = [ 1, 1 ];
var ratios:Array = [ 0, 255 ];
var xMatrix:Matrix = new Matrix();
xMatrix.createGradientBox(radius * 2, radius * 2);
var yMatrix:Matrix = new Matrix();
yMatrix.createGradientBox(radius * 2, radius * 2, Math.PI / 2);

var xCircle:Shape = new Shape();
xCircle.graphics.lineStyle(0, 0, 0);
xCircle.graphics.beginGradientFill(type, redColors, alphas, ratios,
  xMatrix);
xCircle.graphics.drawCircle(radius, radius, radius);

var yCircle:Shape = new Shape();
yCircle.graphics.lineStyle(0, 0, 0);
yCircle.graphics.beginGradientFill(type, blueColors, alphas, ratios,
  yMatrix);
yCircle.graphics.drawCircle(radius, radius, radius);

// 将圆定位在屏幕底部以便于参考。
this.addChild(xCircle);
xCircle.y = stage.stageHeight - xCircle.height;
this.addChild(yCircle);
yCircle.y = stage.stageHeight - yCircle.height;
yCircle.x = 200;
```

```
// 将图像加载到舞台上。
var loader:Loader = new Loader();
var url:URLRequest = new URLRequest("http://www.helpexamples.com/flash/
  images/image1.jpg");
loader.load(url);
this.addChild(loader);

// 通过合并两个渐变圆来创建置换图图像。
var map:BitmapData = new BitmapData(xCircle.width, xCircle.height, false,
  0x7F7F7F);
map.draw(xCircle);
var yMap:BitmapData = new BitmapData(yCircle.width, yCircle.height, false,
  0x7F7F7F);
yMap.draw(yCircle);
map.copyChannel(yMap, yMap.rect, new Point(0, 0), BitmapDataChannel.BLUE,
  BitmapDataChannel.BLUE);
yMap.dispose();

// 在舞台上显示置换图图像以便于参考。
var mapBitmap:Bitmap = new Bitmap(map);
this.addChild(mapBitmap);
mapBitmap.x = 400;
mapBitmap.y = stage.stageHeight - mapBitmap.height;

// 下面的函数在鼠标位置创建置换图滤镜。
function magnify():void
{
  // 定位滤镜。
  var filterX:Number = (loader.mouseX) - (map.width / 2);
  var filterY:Number = (loader.mouseY) - (map.height / 2);
  var pt:Point = new Point(filterX, filterY);
  var xyFilter:DisplacementMapFilter = new DisplacementMapFilter();
  xyFilter.mapBitmap = map;
  xyFilter.mapPoint = pt;
  // 置换图图像中的红色将控制 x 置换。
  xyFilter.componentX = BitmapDataChannel.RED;
  // 置换图图像中的蓝色将控制 y 置换。
  xyFilter.componentY = BitmapDataChannel.BLUE;
  xyFilter.scaleX = 35;
  xyFilter.scaleY = 35;
  xyFilter.mode = DisplacementMapFilterMode.IGNORE;
  loader.filters = [xyFilter];
}

// 移动鼠标时调用下面的函数。如果鼠标位于
// 加载的图像上，则应用滤镜。
function moveMagnifier(event:MouseEvent):void
{
  if (loader.hitTestPoint(loader.mouseX, loader.mouseY))
  {
```

```
        magnify();
    }
}
loader.addEventListener(MouseEvent.MOUSE_MOVE, moveMagnifier);
```

代码首先生成两个渐变圆，它们合并在一起构成置换图图像。红色圆创建 x 轴置换 (`xyFilter.componentX = BitmapDataChannel.RED`)，蓝色圆创建 y 轴置换 (`xyFilter.componentY = BitmapDataChannel.BLUE`)。为帮助您理解置换图图像的外观，代码向屏幕底部添加了原始圆形以及合并后作为置换图图像的圆形。



然后，代码加载一个图像，当鼠标移动时，将置换滤镜应用于鼠标下方的图像部分。用作置换图图像的渐变圆使置换区域从鼠标指针处向外扩展。请注意，置换图图像的灰色区域不会发生置换。灰色为 `0x7F7F7F`。该灰色的蓝色和红色通道与这些颜色通道中的中间色度完全匹配，因此在置换图图像的灰色区域不会发生置换。同样，在圆的中心也不会发生置换。由于蓝色和红色是引起置换的颜色，虽然颜色中没有灰色，但该颜色的蓝色通道和红色通道与中度灰的蓝色通道和红色通道完全相同，因此该处不发生置换。

# 示例：Filter Workbench

Filter Workbench 提供了一个简单的用户界面，用于对图像应用不同的滤镜并查看结果代码，这些代码可用于使用 ActionScript 生成同样的效果。有关此示例和下载源代码的说明，请访问 www.adobe.com/go/learn_fl_filters_cn。

第 16 章

# 处理影片剪辑

16

MovieClip 类是在 Adobe Flash CS3 Professional 中创建的动画和影片剪辑元件的核心类。它除具有显示对象的所有行为和功能外，还具有用于控制影片剪辑的时间轴的其它属性和方法。本章介绍如何使用 ActionScript 来控制影片剪辑回放和动态创建影片剪辑。

## 目录



# 影片剪辑基础知识

## 影片剪辑处理简介

影片剪辑对于使用 Flash 创作工具创建动画内容并想要通过 ActionScript 来控制该内容的人来说是一个重要元素。只要在 Flash 中创建影片剪辑元件，Flash 就会将该元件添加到该 Flash 文档的库中。默认情况下，此元件会成为 MovieClip 类的一个实例，因此具有 MovieClip 类的属性和方法。

在将某个影片剪辑元件的实例放置在舞台上时，如果该影片剪辑具有多个帧，它会自动按其时间轴进行回放，除非使用 ActionScript 更改其回放。此时间轴使 MovieClip 类与其它类区别开来，允许您在 Flash 创作工具中通过补间动画或补间形状来创建动画。相反，对于作为 Sprite 类的实例的显示对象，您只需以编程方式更改该对象的值即可创建动画。

在 ActionScript 的早期版本中，MovieClip 类是舞台上所有实例的基类。在 ActionScript 3.0 中，影片剪辑只是可以在屏幕上显示的众多显示对象中的一个。如果使用显示对象时不需要时间轴，则使用 Shape 类或 Sprite 类替代 MovieClip 类可能会提高呈现性能。有关为任务选择合适的显示对象的详细信息，请参阅第 338 页的 “选择 DisplayObject 子类”。

## 常见的影片剪辑任务

本章介绍以下常见的影片剪辑任务：

- 播放和停止影片剪辑
- 反向播放影片剪辑
- 将播放头移动到影片剪辑时间轴中的特定点
- 在 ActionScript 中处理帧标签
- 在 ActionScript 中访问场景信息
- 使用 ActionScript 创建库影片剪辑元件的实例
- 加载和控制外部 SWF 文件，包括为以前版本的 Flash Player 创建的文件
- 构建一个 ActionScript 系统，用于创建将在运行时加载和使用的图形资源

## 重要概念和术语

以下参考列表包含将会在本章中使用的重要术语：

- AVM1 SWF：使用 ActionScript 1.0 或 ActionScript 2.0 创建的 SWF 文件，通常以 Flash Player 8 或更早版本为目标播放器。
- AVM2 SWF：使用 ActionScript 3.0 for Adobe Flash Player 9 创建的 SWF 文件。
- 外部 SWF：单独从项目 SWF 文件创建的 SWF 文件，将加载到项目 SWF 文件中并在该 SWF 文件中回放。
- 帧：时间轴上划分时间的最小单位。与运动图像电影胶片一样，每个帧都类似于动画在特定时间的快照，当快速按顺序播放各个帧时，会产生动画的效果。
- 时间轴：构成影片剪辑动画序列的一系列帧的比喻形式。 MovieClip 对象的时间轴等同于 Flash 创作工具中的时间轴。
- 播放头：一个标记，用于标识在给定时刻在时间轴中所处的位置 （帧）。

## 完成本章中的示例

学习本章的过程中，您可能想要自己动手测试一些示例代码清单。由于本章是关于在 ActionScript 中处理影片剪辑的，因此，本章中编写的所有代码清单的出发点几乎都是说明如何操作已经创建并放置在舞台上的影片剪辑元件。测试范例将涉及在 Flash Player 中查看结果，以了解代码对元件的影响。要测试本章中的代码清单，请执行以下操作：

1. 创建一个空的 Flash 文档。
2. 在时间轴上选择一个关键帧。
3. 打开 “动作”面板，将代码清单复制到 “脚本”窗格中。

4. 在舞台上创建一个影片剪辑元件实例。例如，绘制一个形状，选中它，选择 “修改” > “转换为元件”，并给元件指定一个名称。
5. 使影片剪辑保持选中，在 “属性” 检查器中为它指定一个实例名称。名称应与示例代码清单中影片剪辑使用的名称相匹配，例如，如果代码清单操作名为 `myMovieClip` 的影片剪辑，则应将影片剪辑实例也命名为 `myMovieClip`。
6. 使用 “控制” > “测试影片” 运行程序。

   在屏幕上，您将看到按照代码清单所指定的要求操作影片剪辑的结果。

测试示例代码清单的其它技术在第 53 页的 “测试本章内的示例代码清单” 中有更详细的介绍。

# 处理 MovieClip 对象

在发布 SWF 文件时，Flash 会将舞台上的所有影片剪辑元件实例转换为 MovieClip 对象。通过在属性检查器的 “实例名称” 字段中指定影片剪辑元件的实例名称，您可以在 ActionScript 中使用该元件。在创建 SWF 文件时，Flash 会生成在舞台上创建该 MovieClip 实例的代码并使用该实例名称声明一个变量。如果您已经命名了嵌套在其它已命名影片剪辑内的影片剪辑，则这些子级影片剪辑将被视为父级影片剪辑的属性，您可以使用点语法访问该子级影片剪辑。例如，如果实例名称为 `childClip` 的影片剪辑嵌套在另一个实例名称为 `parentClip` 的剪辑内，则可以通过调用以下代码来播放子级剪辑的时间轴动画：

```
parentClip.childClip.play()
```

尽管 ActionScript 2.0 MovieClip 类的一些旧方法和属性仍保持不变，但其它方法和属性已发生了变化。所有前缀为下划线的属性均已被重新命名。例如，`_width` 和 `_height` 属性现在分别作为 `width` 和 `height` 被访问，而 `_xscale` 和 `_yscale` 则作为 `scaleX` 和 `scaleY` 被访问。若要查看 MovieClip 类的属性和方法的完整列表，请参阅 《ActionScript 3.0 语言和组件参考》。

# 控制影片剪辑回放

Flash 利用时间轴来形象地表示动画或状态改变。任何使用时间轴的可视元素都必须为 MovieClip 对象或从 MovieClip 类扩展而来。尽管 ActionScript 可控制任何影片剪辑的停止、播放或转至时间轴上的另一点，但不能用于动态创建时间轴或在特定帧添加内容，这项工作仅能使用 Flash 创作工具来完成。

MovieClip 在播放时将以 SWF 文件的帧速率决定的速度沿着其时间轴推进。或者，您也可以通过在 ActionScript 中设置 `Stage.frameRate` 属性来覆盖此设置。

## 播放影片剪辑和停止回放

play() 和 stop() 方法允许对时间轴上的影片剪辑进行基本控制。例如，假设舞台上有一个影片剪辑元件，其中包含一个自行车横穿屏幕的动画，其实例名称设置为 bicycle。如果将以下代码附加到主时间轴上的关键帧，

```
bicycle.stop();
```

自行车将不会移动（将不播放其动画）。自行车的移动可以通过一些其它的用户交互来开始。例如，如果您有一个名为 startButton 的按扭，则主时间轴上某一关键帧上的以下代码会使单击该按扭时播放该动画：

```
// 单击该按扭时调用此函数。它会使
// 自行车动画进行播放。
function playAnimation(event:MouseEvent):void
{
  bicycle.play();
}
// 将该函数注册为按钮的侦听器。
startButton.addEventListener(MouseEvent.CLICK, playAnimation);
```

## 快进和后退

在影片剪辑中，play() 和 stop() 方法并非是控制回放的唯一方法。也可以使用 nextFrame() 和 prevFrame() 方法手动向前或向后沿时间轴移动播放头。调用这两种方法中的任一方法均会停止回放并分别使播放头向前或向后移动一帧。

使用 play() 方法类似于每次触发影片剪辑对象的 enterFrame 事件时调用 nextFrame()。使用此方法，您可以为 enterFrame 事件创建一个事件侦听器并在侦听器函数中让 bicycle 回到前一帧，从而使 bicycle 影片剪辑向后播放，如下所示：

```
// 触发 enterFrame 事件时调用此函数，这意味着
// 每帧调用一次该函数。
function everyFrame(event:Event):void
{
  if (bicycle.currentFrame == 1)
  {
    bicycle.gotoAndStop(bicycle.totalFrames);
  }
  else
  {
    bicycle.prevFrame();
  }
}
bicycle.addEventListener(Event.ENTER_FRAME, everyFrame);
```

在正常回放过程中，如果影片剪辑包含多个帧，播放时将会无限循环播放，也就是说在经过最后一帧后将返回到第 1 帧。使用 prevFrame() 或 nextFrame() 时，不会自动发生此行为（在播放头位于第 1 帧时调用 prevFrame() 不会将播放头移动到最后一帧）。以上示例中的 if 条件将检查播放头是否已返回至第一帧，并将播放头设置为处于最后一帧前面，从而有效地使影片剪辑向后持续循环播放。

## 跳到不同帧和使用帧标签

向新帧发送影片剪辑非常简单。调用 gotoAndPlay() 或 gotoAndStop() 将使影片剪辑跳到指定为参数的帧编号。或者，您可以传递一个与帧标签名称匹配的字符串。可以为时间轴上的任何帧分配一个标签。为此，选择时间轴上的某一帧，然后在属性检查器的 “帧标签”字段中输入一个名称。

当创建复杂的影片剪辑时，使用帧标签比使用帧编号具有明显优势。当动画中的帧、图层和补间的数量变得很大时，应考虑给重要的帧加上具有解释性说明的标签来表示影片剪辑中的行为转换（例如，“离开”、“行走”或 “跑”）。这可提高代码的可读性，同时使代码更加灵活，因为转到指定帧的 ActionScript 调用是指向单一参考（“标签”而不是特定帧编号）的指针。如果您以后决定将动画的特定片段移动到不同的帧，则无需更改 ActionScript 代码，只要将这些帧的相同标签保持在新位置即可。

为便于在代码中表示帧标签，ActionScript 3.0 包括了 FrameLabel 类。此类的每个实例均代表一个帧标签，并具有一个 name 属性 （表示在属性检查器中指定的帧标签的名称）和一个 frame 属性 （表示该标签在时间轴上所处帧的帧编号）。

为了访问与影片剪辑实例相关联的 FrameLabel 实例，MovieClip 类包括了两个可直接返回 FrameLabel 对象的属性。currentLabels 属性返回一个包含影片剪辑整个时间轴上所有 FrameLabel 对象的数组。currentLabel 属性返回一个表示在时间轴上最近遇到的帧标签的 FrameLabel 对象。

假设在创建一个名为 “机器人”的影片剪辑并已经为其动画的各个状态加上了标签。可以设置一个用于检查 currentLabel 属性的条件以访问机器人的当前状态，如以下代码所示：

```
if (robot.currentLabel.name == "walking"
{
    // 完成一些操作
}
```

## 处理场景

在 Flash 创作环境中，您可以使用场景来区分 SWF 文件播放时将要经过的一系列时间轴。使用 `gotoAndPlay()` 或 `gotoAndStop()` 方法的第二个参数，可以指定要向其发送播放头的场景。所有 FLA 文件开始时都只有初始场景，但您可以创建新的场景。

使用场景并非始终是最佳方法，因为场景有许多缺点。包含多个场景的 Flash 文档可能很难维护，尤其是在存在多个作者的环境中。多个场景也会使带宽效率降低，因为发布过程会将所有场景合并为一个时间轴。这样将使所有场景进行渐进式下载，即使从不会播放这些场景。因此，除非是组织冗长的基于多个时间轴的动画，否则通常不鼓励使用多个场景。

MovieClip 类的 `scenes` 属性返回表示 SWF 文件中所有场景的 Scene 对象的数组。`currentScene` 属性返回一个表示当前正在播放的场景的 Scene 对象。

Scene 类具有多个提供有关场景信息的属性。`labels` 属性返回表示该场景中帧标签的 FrameLabel 对象的数组。`name` 属性将以字符串形式返回场景名称。`numFrames` 属性返回一个表示场景中帧的总数的整数。

# 使用 ActionScript 创建 MovieClip 对象

在 Flash 中向屏幕中添加内容的一个方法是将资源从库中拖放到舞台上，但不是仅有这一种方法。对于复杂项目，经验丰富的开发人员通常更喜欢以编程方式创建影片剪辑。这种方法具有多个优点: 代码更易于重用、编译时速度加快，以及仅可在 ActionScript 中进行的更复杂的修改。

ActionScript 3.0 的显示列表 API 简化了动态创建 MovieClip 对象的过程。直接实例化 MovieClip 实例的功能从向显示列表中添加该实例的过程中分离出来，从而更加灵活、简单，而不会牺牲性控制性能。

在 ActionScript 3.0 中，当以编程方式创建影片剪辑 （或任何其它显示对象）实例时，只有通过对显示对象容器调用 `addChild()` 或 `addChildAt()` 方法将该实例添加到显示列表中后，才能在屏幕上看到该实例。这允许您创建影片剪辑、设置其属性，甚至可以在向屏幕呈现该影片剪辑之前调用方法。有关处理显示列表的详细信息，请参阅第 329 页的 “处理显示对象容器”。

## 为 ActionScript 导出库元件

默认情况下， Flash 文档库中的影片剪辑元件实例不能以动态方式创建 （即只使用 ActionScript 创建）。这是由于为在 ActionScript 中使用而导出的每个元件都会增加 SWF 文件的大小，而且众所周知，有些元件可能不适合在舞台上使用。因此，为了使元件可以在 ActionScript 中使用，必须指定为 ActionScript 导出该元件。

**为 ActionScript 导出元件:**

1. 在 “库” 面板中选择该元件并打开其 “元件属性” 对话框。
2. 必要时激活 “高级” 设置。
3. 在 “链接” 部分中，激活 “为 ActionScript 导出” 复选框。

   这将激活 “类” 和 “基类” 字段。

   默认情况下，“类” 字段会用删除空格的元件名称填充 （例如，名为 “Tree House” 的元件会变为 “TreeHouse”）。若要指定该元件对其行为使用自定义类，请在此字段中输入该类的完整名称，包括它所在的包。如果希望能够在 ActionScript 中创建该元件的实例，但不需要添加任何其它行为，则可以使类名称保持原样。

   “基类” 字段的值默认为 `flash.display.MovieClip`。如果想让元件扩展另一个自定义类的功能，可以指定该类的名称替代这一值，只要该类能够扩展 Sprite（或 MovieClip）类即可。
4. 按 “确定” 按钮以保存所做的更改。

   此时，如果 Flash 找不到包含指定类的定义的外部 ActionScript 文件（例如，如果不需要为元件添加其它行为），将显示以下警告：

   *无法在类路径中找到对此类的定义，因此将在导出时自动在 SWF 文件中生成相应的定义。*

   如果库元件不需要超出 MovieClip 类功能的独特功能，则可以忽略此警告消息。

如果没有为元件提供类， Flash 将为元件创建一个等同于下面所示类的类：

```
package
{
  import flash.display.MovieClip;

  public class ExampleMovieClip extends MovieClip
  {
    public function ExampleMovieClip()
    {
    }
  }
}
```

如果想要向元件中添加额外的 ActionScript 功能，请向下面的结构中添加相应的属性和方法。例如，假如有一个包含 50 像素宽和 50 像素高的圆形的影片剪辑元件，并用名为 Circle 的类指定为 ActionScript 导出该元件。以下代码在放入 Circle.as 文件后将扩展 MovieClip 类，同时为此元件提供额外的方法 `getArea()` 和 `getCircumference()`：

```
package
{
  import flash.display.MovieClip;

  public class Circle extends MovieClip
  {
```

```
        public function Circle()
        {
        }

        public function getArea():Number
        {
            // 公式为 Pi 乘以半径的平方。
            return Math.PI * Math.pow((width / 2), 2);
        }

        public function getCircumference():Number
        {
            // 公式为 Pi 乘以直径。
            return Math.PI * width;
        }
    }
}
```

放置在 Flash 文档第 1 帧的关键帧上的以下代码将创建该元件的一个实例，并在屏幕上显示该实例：

```
var c:Circle = new Circle();
addChild(c);
trace(c.width);
trace(c.height);
trace(c.getArea());
trace(c.getCircumference());
```

此代码演示了基于 ActionScript 的实例化可作为将单个资源拖放到舞台上的替代方法。它所创建的圆形具有影片剪辑的所有属性，同时还具有 Circle 类中定义的自定义方法。这是一个非常基本的示例 — 您的库元件可在其类中指定任意数目的属性和方法。

基于 ActionScript 的实例化功能强大，因为允许动态创建大量实例，而如果采用手动方式来创建将是一项繁重的任务。同时还很灵活，因为您可以在创建实例时自定义每个实例的属性。您可以通过使用循环动态创建多个 Circle 实例来体会上述优点。在 Flash 文档库中存在上述 Circle 元件和类的情况下，将以下代码放在第 1 帧的关键帧上：

```
import flash.geom.ColorTransform;

var totalCircles:uint = 10;
var i:uint;
for (i = 0; i < totalCircles; i++)
{
    // 创建一个新的 Circle 实例。
    var c:Circle = new Circle();
    // 将新的 Circle 放在可将圆形在舞台上均匀间隔开
    // 的 x 坐标处。
    c.x = (stage.stageWidth / totalCircles) * i;
    // 将 Circle 实例放在舞台的垂直居中位置。
    c.y = stage.stageHeight / 2;
    // 将 Circle 实例更改为随机颜色
```

```
    c.transform.colorTransform = getRandomColor();
    // 将 Circle 实例添加到当前时间轴。
    addChild(c);
}

function getRandomColor():ColorTransform
{
    // 为红色、绿色和蓝色通道生成随机值。
    var red:Number = (Math.random() * 512) - 255;
    var green:Number = (Math.random() * 512) - 255;
    var blue:Number = (Math.random() * 512) - 255;

    // 使用随机颜色创建并返回 ColorTransform 对象。
    return new ColorTransform(1, 1, 1, 1, red, green, blue, 0);
}
```

此代码演示了如何使用代码快速创建和自定义元件的多个实例。每个实例都根据循环内的当前计数进行定位，并且每个实例都通过设置 `transform` 属性（Circle 通过扩展 MovieClip 类而继承该属性）获得了一种随机颜色。

# 加载外部 SWF 文件

在 ActionScript 3.0 中，SWF 文件是使用 Loader 类来加载的。若要加载外部 SWF 文件，ActionScript 需要执行以下 4 个操作：

1. 用文件的 URL 创建一个新的 URLRequest 对象。
2. 创建一个新的 Loader 对象。
3. 调用 Loader 对象的 `load()` 方法，并以参数形式传递 URLRequest 实例。
4. 对显示对象容器（如 Flash 文档的主时间轴）调用 `addChild()` 方法，将 Loader 实例添加到显示列表中。

最后，代码如下所示：

```
var request:URLRequest = new URLRequest("http://www.[yourdomain].com/
    externalSwf.swf");
var loader:Loader = new Loader()
loader.load(request);
addChild(loader);
```

通过指定图像文件的 URL 而不是 SWF 文件的 URL，可以使用上述同样的代码加载外部图像文件，如 JPEG、GIF 或 PNG 图像。SWF 文件不同于图像文件，可能包含 ActionScript。因此，虽然加载 SWF 文件的过程可能与加载图像的过程完全相同，但如果您计划使用 ActionScript 以某种方式与外部 SWF 文件通信，则在加载该外部 SWF 文件时，执行加载的 SWF 文件和被加载的 SWF 文件必须位于同一个安全沙箱中。另外，如果外部 SWF 文件包含了与执行加载的 SWF 文件中的类共享同一命名空间的类，可能需要为被加载的 SWF 文件创建新的应用程序域才能避免命名空间冲突。有关安全性和应用程序域注意事项的详细信息，请参阅第 603 页的 "使用 ApplicationDomain 类"和第 665 页的 "加载 SWF 文件和图像"。

当成功加载外部 SWF 文件后，可通过 `Loader.content` 属性访问该文件。如果该外部 SWF 文件是针对 ActionScript 3.0 发布的，则加载的文件将为影片剪辑或 sprite，具体取决于所扩展的类。

## 加载早期 SWF 文件的注意事项

如果已使用早期版本的 ActionScript 发布了外部 SWF 文件，则需要考虑一些重要的限制条件。与在 AVM2 (ActionScript Virtual Machine 2) 中运行的 ActionScript 3.0 SWF 文件不同，针对 ActionScript 1.0 或 2.0 发布的 SWF 文件在 AVM1 (ActionScript Virtual Machine 1) 中运行。

成功加载 AVM1 SWF 文件后，已加载的对象（`Loader.content` 属性）将是 AVM1Movie 对象。AVM1Movie 实例不同于 MovieClip 实例。而是显示对象，但不同于影片剪辑，它不包括与时间轴相关的方法或属性。父级 AVM2 SWF 文件将不能访问已加载的 AVM1Movie 对象的属性、方法或对象。

对 AVM2 SWF 文件加载的 AVM1 SWF 文件还有其它限制。有关详细信息，请参阅《ActionScript 3.0 语言和组件参考》中的 AVM1Movie 类列表。

# 示例：RuntimeAssetsExplorer

"为 ActionScript 导出"功能非常适合用于在多个项目间使用库的情况。已经导出到 ActionScript 的元件不仅可用于相应的 SWF 文件，还可用于加载该 SWF 文件的同一安全沙箱内的任何 SWF 文件。这样，单个 Flash 文档可生成仅用于保存图形资源的 SWF 文件。该技术对大型项目来说特别有用，比如使用可视资源的设计人员可以与创建 "包装" SWF 文件然后在运行时加载图形资源 SWF 文件的开发人员同时工作。您可以使用此方法维护一系列图形资源与编程开发进度无关的版本文件。

RuntimeAssetsExplorer 应用程序加载属于 RuntimeAsset 子类的任何 SWF 文件，并允许您浏览该 SWF 文件的可用资源。示例说明了以下过程：

- 使用 `Loader.load()` 加载外部 SWF 文件
- 动态创建为 ActionScript 导出的库元件
- 使用 ActionScript 控制 MovieClip 回放

开始之前，注意每个 SWF 文件必须位于同一个安全沙箱中。有关详细信息，请参阅第 660 页的 “安全沙箱”。

要获取该范例的应用程序文件，请访问 www.adobe.com/go/learn_programmingAS3samples_flash_cn。RuntimeAssetsExplorer 应用程序文件位于文件夹 Samples/RuntimeAssetsExplorer 中。该应用程序包含以下文件：

| 文件 | 说明 |
|---|---|
| RuntimeAssetsExample.mxml<br>或<br>RuntimeAssetsExample.fla | 适用于 Flex (MXML) 或 Flash (FLA) 的应用程序的用户界面。 |
| GeometricAssets.as | 用于实现 RuntimeAsset 接口的示例类。 |
| GeometricAssets.fla | 链接到 GeometricAssets 类的 FLA 文件 （该类是 FLA 的文档类）包含为 ActionScript 导出的元件。 |
| com/example/programmingas3/<br>runtimeassetsexplorer/RuntimeLibrary.as | 用于定义将加载到浏览器容器中的所有运行时资源 SWF 文件所需方法的接口。 |
| com/example/programmingas3/<br>runtimeassetsexplorer/AnimatingBox.as | 形状为旋转方框的库元件的类。 |
| com/example/programmingas3/<br>runtimeassetsexplorer/AnimatingStar.as | 形状为旋转星形的库元件的类。 |

## 建立运行时库界面

若要使浏览器与 SWF 库正确交互，运行时资源库的结构就必须具有一定程式。我们将通过创建一个接口（在用于区分预期结构的方法蓝图方面，它类似于一个类，但它又不像是一个类，因为它不包括方法体）来完成此过程。接口提供了在运行时库和浏览器之间相互通信的方法。加载到浏览器中的运行时资源的每个 SWF 都将实现此接口。有关接口和如何使用接口的详细信息，请参阅第 131 页的 “接口”。

RuntimeLibrary 接口非常简单 — 我们只需要能够为浏览器提供类路径数组以导出元件并使元件在运行时库中可用的函数。为此，该接口具有单个方法：`getAssets()`。

```
package com.example.programmingas3.runtimeassetexplorer
{
    public interface RuntimeLibrary
    {
```

```
    function getAssets():Array;
  }
}
```

# 创建资源库 SWF 文件

通过定义 RuntimeLibrary 界面，可以创建能够加载到另一个 SWF 文件中的多个资源库 SWF 文件。制作资源的单个 SWF 库包括四个任务：

- 为资源库 SWF 文件创建一个类
- 为库中包含的单个资源创建类
- 创建实际图形资源
- 将图形元素与类关联并发布库 SWF

## 创建一个类以实现 RuntimeLibrary 接口

下一步，我们将创建用于实现 RuntimeLibrary 接口的 GeometricAssets 类。它将是 FLA 的文档类。此类的代码与 RuntimeLibrary 接口非常相似，它们之间的不同在于，在类定义中，`getAssets()` 方法具有方法体。

```
package
{
  import flash.display.Sprite;
  import com.example.programmingas3.runtimeassetexplorer.RuntimeLibrary;

  public class GeometricAssets extends Sprite implements RuntimeLibrary
  {
    public function GeometricAssets() {

    }
    public function getAssets():Array {
      return [
  "com.example.programmingas3.runtimeassetexplorer.AnimatingBox",
          "com.example.programmingas3.runtimeassetexplorer.AnimatingStar"
  ];
    }
  }
}
```

如果要创建第二个运行时库，可以创建另一个基于其它类（例如 AnimationAssets）的 FLA，该类可提供自带的 `getAssets()` 实现。

## 为每个 MovieClip 资源创建类

对于本示例，我们将只扩展 MovieClip 类而不对自定义资源添加任何功能。以下 AnimatingStar 的代码类似于 AnimatingBox 的代码：

```
package com.example.programmingas3.runtimeassetexplorer
{
  import flash.display.MovieClip;

  public class AnimatingStar extends MovieClip
  {
    public function AnimatingStar() {
    }
  }
}
```

## 发布库

现在将基于 MovieClip 的资源连接到新类，方法是创建一个新的 FLA 并在"属性"检查器的"文档类"字段中输入 GeometricAssets。为达到本示例的目的，我们将创建两个使用时间轴补间的非常基本的形状，其中一个形状顺时针旋转超过 360 帧。`animatingBox` 和 `animatingStar` 元件都设为"为 ActionScript 导出"，并将"类"字段设置为 `getAssets()` 实现中指定的对应类路径。保留 `flash.display.MovieClip` 的默认基类，因为我们希望对标准 MovieClip 方法进行子分类。

在设置完元件导出设置后，发布 FLA。现在您便拥有第一个运行时库了。该 SWF 文件可以加载到另一个 AVM2 SWF 文件中，且 AnimatingBox 和 AnimatingStar 元件可用于新的 SWF 文件。

# 将库加载到另一个 SWF 文件中

要处理的最后一个功能性部分是资源浏览器的用户界面。在本示例中，运行时库的路径硬编码为一个名为 `ASSETS_PATH` 的变量。或者，您也可以使用 FileReference 类。例如，创建浏览以查找硬盘上特定 SWF 文件的界面。

成功加载运行时库后， Flash Player 会调用 `runtimeAssetsLoadComplete()` 方法：

```
private function runtimeAssetsLoadComplete(event:Event):void
{
  var rl:* = event.target.content;
  var assetList:Array = rl.getAssets();
  populateDropdown(assetList);
  stage.frameRate = 60;
}
```

在此方法中，变量 rl 表示已加载的 SWF 文件。代码将调用已加载的 SWF 文件的 getAssets() 方法，获取可用资源的列表，并通过调用 populateDropDown() 方法用这些资源填充具有可用资源列表的 ComboBox 组件。该方法会依次存储每个资源的完整类路径。单击用户界面上的 “添加” 按扭即会触发 addAsset() 方法：

```
private function addAsset():void
{
    var className:String = assetNameCbo.selectedItem.data;
    var AssetClass:Class = getDefinitionByName(className) as Class;
    var mc:MovieClip = new AssetClass();
    ...
}
```

此方法将获取 ComboBox 中当前所选资源的类路径 (assetNameCbo.selectedItem.data)，并使用 getDefinitionByName() 函数（来自 flash.utils 包）获取对该资源的类的实际引用，以创建该资源的新实例。

第 17 章

# 处理文本

17

在 ActionScript 3.0 中，文本通常是在文本字段之内显示，但是偶尔也会作为项目的属性出现在显示列表中 （例如，作为某个 UI 组件的标签）。本章介绍如何使用由脚本定义的文本字段内容，以及如何使用用户输入、来自远程文件的动态文本或在 Adobe Flash CS3 Professional 中定义的静态文本。作为 ActionScript 3.0 编程人员，您可以指定文本字段的具体内容，或指定文本源，然后使用样式和格式设置该文本的外观。还可以在用户输入文本或单击超链接时响应用户事件。

## 目录

# 处理文本的基础知识

## 处理文本简介

在 Adobe Flash Player 中，若要在屏幕上显示文本，可以使用 TextField 类的实例。TextField 类是 Adobe Flex 框架和 Flash 创作环境中提供的其它基于文本的组件（如 TextArea 组件或 TextInput 组件）的基础。有关在 Flash 创作环境中使用文本组件的详细信息，请参阅《使用 Flash》的 “关于文本控制”。

文本字段内容可以在 SWF 文件中预先指定、从外部源（如文本文件或数据库）中加载或由用户在与应用程序交互时输入。在文本字段内，文本可以显示为呈现的 HTML 内容，并可在其中嵌入图像。一旦建立了文本字段的实例，您可以使用 flash.text 包中的类（例如 TextFormat 类和 StyleSheet 类）来控制文本的外观。flash.text 包几乎包含与在 ActionScript 中创建文本、管理文本及对文本进行格式设置有关的所有类。

可以用 TextFormat 对象定义格式设置并将此对象分配给文本字段，以此来设置文本格式。如果文本字段包含 HTML 文本，则可以对文本字段应用 StyleSheet 对象，以便将样式分配给文本字段内容的特定片段。TextFormat 对象或 StyleSheet 对象包含定义文本外观（例如颜色、大小和粗细）的属性。TextFormat 对象可以将属性分配给文本字段中的所有内容，也可以分配给某个范围的文本。例如，在同一文本字段中，一个句子可以是粗体的红色文本，而下一个句子可以是斜体的蓝色文本。

有关文本格式的详细信息，请参阅第 449 页的 “指定文本格式”。

有关文本字段中 HTML 文本的详细信息，请参阅第 443 页的 “显示 HTML 文本”。

有关样式表的详细信息，请参阅第 450 页的 “应用层叠样式表”。

除了 flash.text 包中的类以外，您还可以使用 flash.events.TextEvent 类响应与文本相关的用户操作。

## 处理文本的常见任务

本章介绍以下与文本相关的常见任务：

- 修改文本字段内容
- 在文本字段中使用 HTML
- 在文本字段中使用图像
- 选择文本并处理用户选择的文本
- 捕获文本输入
- 限制文本输入
- 对文本应用格式设置和 CSS 样式

- 使用清晰度、粗细和消除锯齿功能来微调文本显示
- 在 ActionScript 中访问和处理静态文本字段

## 重要概念和术语

以下参考列表包含您将会在本章中遇到的重要术语：

- 层叠样式表 (Cascading style sheet)：对以 XML（或 HTML）格式构成的内容指定样式和格式设置的标准语法。
- 设备字体 (Device font)：安装在用户计算机上的字体。
- 动态文本字段 (Dynamic text field)：可以由 ActionScript 更改内容而不能由用户输入内容的文本字段。
- 嵌入字体 (Embedded font)：字符轮廓数据存储在应用程序的 SWF 文件中的一种字体。
- HTML 文本：使用 ActionScript 输入到文本字段中的文本内容，包括 HTML 格式标签和实际文本内容。
- 输入文本字段 (Input text field)：其内容可通过用户输入也可以通过 ActionScript 进行更改的文本字段。
- 静态文本字段 (Static text field)：在 Flash 创作工具中创建的文本字段，运行 SWF 文件时不能更改其内容。
- 文本行量度 (Text line metri)：文本字段中文本内容不同部分的大小的量度，如文本的基线、字符顶部的高度、下行字符 （某些小写字母延伸到基线以下的部分）的大小等等。

## 完成本章中的示例

学习本章的过程中，您可能想要自己动手测试一些示例代码清单。由于本章是有关在 ActionScript 中处理文本字段的，因此，本章中几乎所有代码清单都涉及对 TextField 对象（可能是在 Flash 创作工具中创建并放置在舞台上的对象，也可能是使用 ActionScript 创建的对象）的操作。测试范例将涉及在 Flash Player 中查看结果，以了解代码对文本字段的影响。

本章中的示例分为两组。一种类型的示例操作 TextField 对象而不显式创建该对象。要测试本章中的这些代码清单，请执行以下操作：

1. 创建一个空的 Flash 文档。
2. 在时间轴上选择一个关键帧。
3. 打开 “动作”面板，将代码清单复制到 “脚本”窗格中。
4. 使用 “文本”工具在舞台上创建一个动态文本字段。
5. 使文本字段保持选中，在 “属性”检查器中为它指定一个实例名称。名称应与示例代码清单中文本字段使用的名称相匹配，例如，如果代码清单操作名为 `myTextField` 的文本字段，则应将文本字段也命名为 `myTextField`。

6. 使用 “控制” > “测试影片”运行程序。

   在屏幕上，您将看到按照代码清单所指定的要求操作文本字段的结果。

本章中另一个类型的示例代码清单包含一个旨在用作 SWF 的文档类的类定义。在这些列表中，TextField 实例由示例代码创建，因此您不需要单独创建。要测试此类型的代码清单，请执行以下操作：

1. 创建一个空的 Flash 文档并将它保存到您的计算机上。
2. 创建一个新的 ActionScript 文件，并将它保存到 Flash 文档所在的目录中。文件名应与代码清单中的类的名称一致。例如，如果代码清单定义一个名为 TextFieldTest 的类，则使用名称 TextFieldTest.as 来保存 ActionScript 文件。
3. 将代码清单复制到 ActionScript 文件中并保存该文件。
4. 在 Flash 文档中，单击舞台或工作区的空白部分，以激活文档的 “属性”检查器。
5. 在“属性”检查器的“文档类”字段中，输入您从文本中复制的 ActionScript 类的名称。
6. 使用 “控制” > “测试影片”运行程序。

   您将在屏幕上看到示例的结果。

测试示例代码清单的其它技术在第 53 页的“测试本章内的示例代码清单”中有更详细的介绍。

# 显示文本

尽管诸如 Adobe Flex Builder 和 Flash 等创作工具为显示文本提供了几种不同的选择 （包括与文本相关的组件或文本工具），但以编程方式显示文本的主要途径还是通过文本字段。

## 文本类型

文本字段中文本的类型根据其来源进行划分：

■ 动态文本

  动态文本包含从外部源（例如文本文件、XML 文件以及远程 Web 服务）加载的内容。有关详细信息，请参阅第 442 页的 “文本类型”。

■ 输入文本

  输入文本是指用户输入的任何文本或用户可以编辑的动态文本。可以设置样式表来设置输入文本的格式，或使用 flash.text.TextFormat 类为输入内容指定文本字段的属性。有关详细信息，请参阅第 447 页的 “捕获文本输入”。

■ 静态文本

  静态文本只能通过 Flash 创作工具来创建。您无法使用 ActionScript 3.0 创建静态文本实例。但是，可以使用 ActionScript 类（例如 StaticText 和 TextSnapshot）来操作现有的静态文本实例。有关详细信息，请参阅第 455 页的 “处理静态文本”。

## 修改文本字段内容

可以通过将一个字符串赋予 flash.text.TextField.text 属性来定义动态文本。可以直接将字符串赋予该属性，如下所示：

```
myTextField.text = “Hello World”;
```

还可以为 text 属性赋予在脚本中定义的变量值，如下例所示：

```
package
{
  import flash.display.Sprite;
  import flash.text.*;

  public class TextWithImage extends Sprite
  {
    private var myTextBox:TextField = new TextField();
    private var myText:String = "Hello World";

    public function TextWithImage()
    {
      addChild(myTextBox);
      myTextBox.text = myText;
    }
  }
}
```

或者，可以将一个远程变量的值赋予 text 属性。从远程源加载文本值有三种方式：

- flash.net.URLLoader 和 flash.net.URLRequest 类可以从本地或远程位置为文本加载变量。
- FlashVars 属性被嵌入到承载 SWF 文件的 HTML 页中，可以包含文本变量的值。
- flash.net.SharedObject 类管理值的永久存储。有关详细信息，请参阅第 576 页的 “存储本地数据”。

## 显示 HTML 文本

flash.text.TextField 类具有一个 htmlText 属性，可使用它将您的文本字符串标识为包含用于设置内容格式的 HTML 标签。如下例所示，必须将您的字符串值赋予 htmlText 属性（而不是 text 属性），以便 Flash Player 将文本呈现为 HTML：

```
var myText:String = "<p>This is <b>some</b> content to <i>render</i> as
  <u>HTML</u> text.</p>";
myTextBox.htmlText = myText;
```

Flash Player 支持可用于 htmlText 属性的 HTML 标签和实体的一个子集。《ActionScript 3.0 语言和组件参考》中的 flash.text.TextField.htmlText 属性说明提供了有关受支持的 HTML 标签和实体的详细信息。

一旦您使用 htmlText 属性指定了内容，就可以使用样式表或 textformat 标签来管理内容的格式设置。有关详细信息，请参阅第 449 页的 “设置文本格式”。

## 在文本字段中使用图像

将内容显示为 HTML 文本的另一个好处是可以在文本字段中包括图像。可以使用 img 标签引用一个本地或远程图像，并使其显示在关联的文本字段内。

以下示例将创建一个名为 myTextBox 的文本字段，并在显示的文本中包括一个内容为眼睛的 JPG 图像，该图像与 SWF 文件存储在同一目录下：

```
package
{
  import flash.display.Sprite;
  import flash.text.*;

  public class TextWithImage extends Sprite
  {
    private var myTextBox:TextField;
    private var myText:String = "<p>This is <b>some</b> content to <i>test</
  i> and <i>see</i></p><p><img src='eye.jpg' width='20' height='20'></
  p><p>what can be rendered.</p><p>You should see an eye image and some
  <u>HTML</u> text.</p>";

    public function TextWithImage()
    {
      myTextBox.width = 200;
      myTextBox.height = 200;
      myTextBox.multiline = true;
      myTextBox.wordWrap = true;
      myTextBox.border = true;

      addChild(myTextBox);
      myTextBox.htmlText = myText;
    }
  }
}
```

img 标签支持 JPEG、GIF、PNG 和 SWF 文件。

## 在文本字段中滚动文本

在许多情况下，文本会比显示该文本的文本字段长。或者，某个输入字段允许用户输入比字段一次可显示的文本内容更多的文本。您可以使用 flash.text.TextField 类的与滚动相关的属性来管理过长的内容 （垂直或水平方向）。

与滚动有关的属性包括 TextField.scrollV、TextField.scrollH、maxScrollV 和 maxScrollH。可使用这些属性来响应鼠标单击或按键等事件。

以下示例将创建一个已设置大小的文本字段，其中包含的文本内容超过了该字段一次可以显示的内容。用户单击文本字段时，文本会在垂直方向上滚动。

```
package
{
  import flash.display.Sprite;
  import flash.text.*;
  import flash.events.MouseEvent;

  public class TextScrollExample extends Sprite
  {
    private var myTextBox:TextField = new TextField();
    private var myText:String = "Hello world and welcome to the show. It's
  really nice to meet you. Take your coat off and stay a while. OK, show is
  over. Hope you had fun. You can go home now. Don't forget to tip your
  waiter. There are mints in the bowl by the door. Thank you. Please come
  again.";

    public function TextScrollExample()
    {
      myTextBox.text = myText;
      myTextBox.width = 200;
      myTextBox.height = 50;
      myTextBox.multiline = true;
      myTextBox.wordWrap = true;
      myTextBox.background = true;
      myTextBox.border = true;

      var format:TextFormat = new TextFormat();
      format.font = "Verdana";
      format.color = 0xFF0000;
      format.size = 10;

      myTextBox.defaultTextFormat = format;
      addChild(myTextBox);
      myTextBox.addEventListener(MouseEvent.MOUSE_DOWN, mouseDownScroll);
    }

    public function mouseDownScroll(event:MouseEvent):void
    {
      myTextBox.scrollV++;
    }
  }
}
```

# 选择和操作文本

您可以选择动态文本或输入文本。由于 TextField 类的文本选择属性和方法使用索引位置来设置要操作的文本的范围，因此即使不知道内容，您也可以以编程方式选择动态文本或输入文本。



在 Flash 创作工具中，如果对静态文本字段选择了可选选项，则导出并置于显示列表中的文本字段为常规的动态文本字段。

## 选择文本

默认情况下，`flash.text.TextField.selectable` 属性为 `true`，您可以使用 `setSelection()` 方法以编程方式选择文本。

例如，您可以将某个文本字段中的特定文本设置成用户单击该文本字段时处于选定状态：

```
var myTextField:TextField = new TextField();
myTextField.text = "No matter where you click on this text field the TEXT IN
  ALL CAPS is selected.";
myTextField.autoSize = TextFieldAutoSize.LEFT;
addChild(myTextField);
addEventListener(MouseEvent.CLICK, selectText);

function selectText(event:MouseEvent):void
{
  myTextField.setSelection(49, 65);
}
```

同样，如果您想让文本字段中的文本一开始显示时就处于选定状态，可以创建一个在向显示列表中添加该文本字段时调用的事件处理函数。

## 捕获用户选择的文本

TextField 类的 `selectionBeginIndex` 和 `selectionEndIndex` 属性可用于捕获用户当前选择的内容，这两个属性为 “只读”属性，因此不能设置为以编程方式选择文本。此外，输入文本字段也可以使用 `caretIndex` 属性。

例如，以下代码将跟踪用户所选文本的索引值：

```
var myTextField:TextField = new TextField();
myTextField.text = "Please select the TEXT IN ALL CAPS to see the index
  values for the first and last letters.";
myTextField.autoSize = TextFieldAutoSize.LEFT;
addChild(myTextField);
addEventListener(MouseEvent.MOUSE_UP, selectText);

function selectText(event:MouseEvent):void
{
```

```
  trace("First letter index position: " + myTextField.selectionBeginIndex);
  trace("Last letter index position: " + myTextField.selectionEndIndex);
}
```

您可以对所选内容应用 TextFormat 对象属性的集合来更改文本的外观。有关对所选文本应用 TextFormat 属性集合的详细信息，请参阅第 452 页的 "设置文本字段内文本范围的格式"。

# 捕获文本输入

默认情况下，文本字段的 `type` 属性设置为 `dynamic`。如果使用 TextFieldType 类将 `type` 属性设置为 `input`，则可以收集用户输入并保存该值以便在应用程序的其它部分使用。对于表单以及希望用户定义可用于程序中其它位置的文本值的任何应用程序而言，输入文本字段都十分有用。

例如，以下代码会创建一个名为 `myTextBox` 的输入文本字段。当用户在字段中输入文本时，会触发 `textInput` 事件。名为 `textInputCapture` 的事件处理函数会捕获输入的文本字符串，并将其赋予一个变量。Flash Player 会在另一个名为 `myOutputBox` 的文本字段中显示新文本。

```
package
{
  import flash.display.Sprite;
  import flash.display.Stage;
  import flash.text.*;
  import flash.events.*;

  public class CaptureUserInput extends Sprite
  {
    private var myTextBox:TextField = new TextField();
    private var myOutputBox:TextField = new TextField();
    private var myText:String = "Type your text here.";

    public function CaptureUserInput()
    {
      captureText();
    }

    public function captureText():void
    {
      myTextBox.type = TextFieldType.INPUT;
      myTextBox.background = true;
      addChild(myTextBox);
      myTextBox.text = myText;
      myTextBox.addEventListener(TextEvent.TEXT_INPUT, textInputCapture);
    }
```

```
    public function textInputCapture(event:TextEvent):void
    {
      var str:String = myTextBox.text;
      createOutputBox(str);
    }

    public function createOutputBox(str:String):void
    {
      myOutputBox.background = true;
      myOutputBox.x = 200;
      addChild(myOutputBox);
      myOutputBox.text = str;
    }

  }
}
```

# 限制文本输入

由于输入文本字段经常用于表单或应用程序中的对话框，因此您可能想要限制用户在文本字段中输入的字符的类型，或者甚至想将文本隐藏 （例如，文本为密码）。可以设置 flash.text.TextField 类的 `displayAsPassword` 属性和 `restrict` 属性来控制用户输入。

`displayAsPassword` 属性只是在用户键入文本时将其隐藏 （显示为一系列星号）。当 `displayAsPassword` 设置为 `true` 时，"剪切" 和 "复制" 命令及其对应的键盘快捷键将不起作用。如下例所示，为 `displayAsPassword` 属性赋值的过程与为其它属性 （如背景和颜色）赋值类似：

```
myTextBox.type = TextFieldType.INPUT;
myTextBox.background = true;
myTextBox.displayAsPassword = true;
addChild(myTextBox);
```

`restrict` 属性则更为复杂一些，因为您需要指定允许用户在输入文本字段中键入哪些字符。可以允许特定字母、数字或字母、数字和字符的范围。以下代码只允许用户在文本字段中输入大写字母 （不包括数字或特殊字符）：

```
myTextBox.restrict = “A-Z”;
```

ActionScript 3.0 使用连字符来定义范围，使用尖号来定义被排除的字符。有关定义输入文本字段中受限制的内容的详细信息，请参阅 《ActionScript 3.0 语言和组件参考》中的 `flash.text.TextField.restrict` 属性条目。

# 设置文本格式

以编程方式设置文本显示的格式设置有多种方式。可以直接在 TextField 实例中设置属性，例如，`TextFIeld.thickness`、`TextField.textColor` 和 `TextField.textHeight` 属性。也可以使用 `htmlText` 属性指定文本字段的内容，并使用受支持的 HTML 标签，如 `b`、`i` 和 `u`。但是您也可以将 TextFormat 对象应用于包含纯文本的文本字段，或将 StyleSheet 对象应用于包含 `htmlText` 属性的文本字段。使用 TextFormat 和 StyleSheet 对象可以对整个应用程序的文本外观提供最有力的控制和最佳的一致性。可以定义 TextFormat 或 StyleSheet 对象并将其应用于应用程序中的部分或所有文本字段。

## 指定文本格式

您可以使用 TextFormat 类设置多个不同的文本显示属性，并将它们应用于 TextField 对象的整个内容或一定范围的文本。

以下示例对整个 TextField 对象应用一个 TextFormat 对象，并对 TextField 对象中一定范围的文本应用另一个 TextFormat 对象：

```
var tf:TextField = new TextField();
tf.text = "Hello Hello";

var format1:TextFormat = new TextFormat();
format1.color = 0xFF0000;

var format2:TextFormat = new TextFormat();
format2.font = "Courier";

tf.setTextFormat(format1);
var startRange:uint = 6;
tf.setTextFormat(format2, startRange);

addChild(tf);
```

`TextField.setTextFormat()` 方法只影响已显示在文本字段中的文本。如果 TextField 中的内容发生更改，则应用程序可能需要重新调用 `TextField.setTextFormat()` 方法以便重新应用格式设置。您也可以设置 TextField 对象的 `defaultTextFormat` 属性来指定要用于用户输入文本的格式。

## 应用层叠样式表

文本字段可以包含纯文本或 HTML 格式的文本。纯文本存储在实例的 `text` 属性中，而 HTML 文本存储在 `htmlText` 属性中。

您可以使用 CSS 样式声明来定义可应用于多种不同文本字段的文本样式。CSS 样式声明可以在应用程序代码中进行创建，也可以在运行时从外部 CSS 文件中加载。

flash.text.StyleSheet 类用于处理 CSS 样式。 StyleSheet 类可识别有限的 CSS 属性集合。有关 StyleSheet 类支持的样式属性的详细列表，请参阅 《ActionScript 3.0 语言和组件参考》中的 flash.textStylesheet 条目。

如以下示例所示，您可以在代码中创建 CSS，并使用 StyleSheet 对象对 HTML 文本应用这些样式：

```
var style:StyleSheet = new StyleSheet();

var styleObj:Object = new Object();
styleObj.fontSize = "bold";
styleObj.color = "#FF0000";
style.setStyle(".darkRed", styleObj);

var tf:TextField = new TextField();
tf.styleSheet = style;
tf.htmlText = "<span class = 'darkRed'>Red</span> apple";

addChild(tf);
```

创建 StyleSheet 对象后，示例代码创建一个简单对象以容纳一组样式声明属性。之后，代码将调用 `StyleSheet.setStyle()` 方法，将名为 “.darkred” 的新样式添加到样式表中。然后，代码通过将 StyleSheet 对象分配给 TextField 对象的 `styleSheet` 属性来应用样式表格式。

要使 CSS 样式生效，应在设置 `htmlText` 属性之前对 TextField 对象应用样式表。

根据设计，带有样式表的文本字段是不可编辑的。如果您有一个输入文本字段并为其分配一个样式表，则该文本字段将显示样式表的属性，但不允许用户在其中输入新的文本。而且，您也无法在分配有样式表的文本字段上使用以下 ActionScript API：

- `TextField.replaceText()` 方法
- `TextField.replaceSelectedText()` 方法
- `TextField.defaultTextFormat` 属性
- `TextField.setTextFormat()` 方法

如果某个文本字段已经分配了一个样式表，但后来将 `TextField.styleSheet` 属性设置为 `null`，则 `TextField.text` 和 `TextField.htmlText` 属性的内容会向它们的内容中添加标签和属性，以结合先前分配的样式表设定的格式。若要保留原始 `htmlText` 属性，应在将样式表设置为 `null` 之前将其保存在变量中。

## 加载外部 CSS 文件

用于设置格式的 CSS 方法的功能更加强大，您可以在运行时从外部文件加载 CSS 信息。当 CSS 数据位于应用程序本身以外时，您可以更改应用程序中的文本的可视样式，而不必更改 ActionScript 3.0 源代码。部署完应用程序后，可以通过更改外部 CSS 文件来更改应用程序的外观，而不必重新部署应用程序的 SWF 文件。

`StyleSheet.parseCSS()` 方法可将包含 CSS 数据的字符串转换为 StyleSheet 对象中的样式声明。以下示例显示如何读取外部 CSS 文件并对 TextField 对象应用其样式声明。

首先，下面是要加载的 CSS 文件（名为 example.css）的内容：

```
p {
  font-family: Times New Roman, Times, _serif;
  font-size: 14;
}

h1 {
  font-family: Arial, Helvetica, _sans;
  font-size: 20;
  font-weight: bold;
}

.bluetext {
  color: #0000CC;
}
```

接下来是加载该 example.css 文件并对 TextField 内容应用样式的类的 ActionScript 代码：

```
package
{
  import flash.display.Sprite;
  import flash.events.Event;
  import flash.net.URLLoader;
  import flash.net.URLRequest;
  import flash.text.StyleSheet;
  import flash.text.TextField;
  import flash.text.TextFieldAutoSize;

  public class CSSFormattingExample extends Sprite
  {
    var loader:URLLoader;
    var field:TextField;
    var exampleText:String = "<h1>This is a headline</h1>" +
      "<p>This is a line of text. <span class='bluetext'>" +
      "潘his line of text is colored blue.</span></p>";

    public function CSSFormattingExample():void
    {
      field = new TextField();
      field.width = 300;
```

```
      field.autoSize = TextFieldAutoSize.LEFT;
      field.wordWrap = true;
      addChild(field);

      var req:URLRequest = new URLRequest("example.css");

      loader = new URLLoader();
      loader.addEventListener(Event.COMPLETE, onCSSFileLoaded);
      loader.load(req);
    }

    public function onCSSFileLoaded(event:Event):void
    {
      var sheet:StyleSheet = new StyleSheet();
      sheet.parseCSS(loader.data);
      field.styleSheet = sheet;
      field.htmlText = exampleText;
    }
  }
}
```

加载 CSS 数据后，会执行 `onCSSFileLoaded()` 方法并调用 `StyleSheet.parseCSS()` 方法，将样式声明传送到 StyleSheet 对象。

## 设置文本字段内文本范围的格式

flash.text.TextField 类的一个特别有用的方法是 `setTextFormat()` 方法。使用 `setTextFormat()`，您可以将特定属性分配给文本字段的部分内容以响应用户输入，例如，需要提醒用户必须输入特定条目的表单，或在用户选择部分文本时提醒用户更改文本字段内文本段落小节的重点的表单。

以下示例对某一范围的字符使用 `TextField.setTextFormat()`，以便在用户单击文本字段时更改 `myTextField` 的部分内容的外观：

```
var myTextField:TextField = new TextField();
myTextField.text = "No matter where you click on this text field the TEXT IN
  ALL CAPS changes format.";
myTextField.autoSize = TextFieldAutoSize.LEFT;
addChild(myTextField);
addEventListener(MouseEvent.CLICK, changeText);

var myformat:TextFormat = new TextFormat();
myformat.color = 0xFF0000;
myformat.size = 18;
myformat.underline = true;

function changeText(event:MouseEvent):void
{
  myTextField.setTextFormat(myformat, 49, 65);
}
```

# 高级文本呈现

ActionScript 3.0 在 flash.text 包中提供多个类来控制所显示文本的属性，包括嵌入字体、消除锯齿设置、alpha 通道控制及其它特定设置。《ActionScript 3.0 语言和组件参考》提供了有关这些类和属性（包括 CSMSettings、 Font 和 TextRenderer 类）的详细说明。

## 使用嵌入字体

您在应用程序中为 TextField 指定特定字体时， Flash Player 会查找具有相同名称的设备字体（位于用户计算机上的字体）。如果在用户系统上没有找到该字体，或者如果用户的字体版本与具有该名称的字体略有差异，则文本显示外观会与预想的情况差别很大。

若要确保用户看到完全正确的字体，您可以将该字体嵌入到应用程序的 SWF 文件中。嵌入字体有很多好处：

- 嵌入字体字符是消除锯齿的，特别是对于较大的文本，该字体可以使文本边缘看起来更平滑。
- 可以旋转使用嵌入字体的文本。
- 嵌入字体文本可以产生透明或半透明效果。
- 可以对嵌入字体使用字距调整的 CSS 样式。

使用嵌入字体的最大限制是嵌入字体会增加文件大小或应用程序的下载大小。

将声音文件嵌入到应用程序的 SWF 文件中的具体方法因开发环境而异。

嵌入字体后，可以确保 TextField 使用正确的嵌入字体：

- 将 TextField 的 `embedFonts` 属性设置为 `true`。
- 创建一个 TextFormat 对象，将其 `fontFamily` 属性设置为嵌入字体的名称，并对 TextField 应用 TextFormat 对象。指定嵌入字体时，`fontFamily` 属性应只包含一个名称；该名称不能是用逗号分隔的由多个字体名称构成的列表。
- 如果使用 CSS 样式为 TextField 或组件设置字体，请将 `font-family` CSS 属性设置为嵌入字体的名称。如果要指定一种嵌入字体，则 `font-family` 属性必须包含单一名称，而不能是多个名称的列表。

## 在 Flash 中嵌入字体

Flash 创作工具可嵌入系统中已安装的几乎所有字体，包括 TrueType 字体和 Type 1 Postscript 字体。

有多种在 Flash 应用程序中嵌入字体的方法，包括：

- 在舞台上设置 TextField 的字体和样式属性，然后单击 “嵌入字体”复选框
- 创建并引用字体元件
- 创建并使用包含嵌入字体元件的运行时共享库

有关如何在 Flash 应用程序中嵌入字体的详细信息，请参阅《使用 Flash》中的 “动态或输入文本字段的嵌入字体”。

# 控制清晰度、粗细和消除锯齿

默认情况下，在文本调整大小、更改颜色或在不同背景上显示时， Flash Player 可以确定文本显示控件的设置 （如清晰度、粗细和消除锯齿）。在某些情况下，如文本很小、很大或显示在各种特别的背景上时，您可能需要保持您自己对这些设置的控制。可以使用 `flash.text.TextRenderer` 类及其相关类 （如 CSMSettings 类）来覆盖 Flash Player 设置。使用这些类可以精确控制嵌入文本的呈现品质。有关嵌入字体的详细信息，请参阅第 453 页的 “使用嵌入字体”。



`flash.text.TextField.antiAliasType` 属性必须具有 `AntiAliasType.ADVANCED` 值（该值为默认值），以供您设置清晰度、粗细或 `gridFitType` 属性，或供您使用 `TextRenderer.setAdvancedAntiAliasingTable()` 方法。

以下示例使用名为 `myFont` 的嵌入字体对显示的文本应用自定义连续笔触调制 (CSM) 属性和格式设置。用户单击显示的文本时， Flash Player 会应用自定义设置：

```
var format:TextFormat = new TextFormat();
format.color = 0x336699;
format.size = 48;
format.font = "myFont";

var myText:TextField = new TextField();
myText.embedFonts = true;
myText.autoSize = TextFieldAutoSize.LEFT;
myText.antiAliasType = AntiAliasType.ADVANCED;
myText.defaultTextFormat = format;
myText.selectable = false;
myText.mouseEnabled = true;
myText.text = "Hello World";
addChild(myText);
myText.addEventListener(MouseEvent.CLICK, clickHandler);

function clickHandler(event:Event):void
```

```
{
  var myAntiAliasSettings = new CSMSettings(48, 0.8, -0.8);
  var myAliasTable:Array = new Array(myAntiAliasSettings);
  TextRenderer.setAdvancedAntiAliasingTable("myFont", FontStyle.ITALIC,
  TextColorType.DARK_COLOR, myAliasTable);
}
```

# 处理静态文本

静态文本只能在 Flash 创作工具中创建。不能使用 ActionScript 以编程方式对静态文本进行实例化。静态文本用于比较短小并且不会更改（而动态文本则会更改）的文本。可以将静态文本看作类似于在 Flash 创作工具中在舞台上绘制的圆或正方形的一种图形元素。由于静态文本比动态文本受到更多的限制， ActionScript 3.0 不支持使用 flash.text.StaticText 类读取静态文本属性值的能力。另外，您可以使用 flash.text.TextSnapshot 类从静态文本中读取值。

## 使用 StaticText 类访问静态文本字段

通常，可以在 Flash 创作工具的 "动作" 面板中使用 flash.text.StaticText 类来与放置在舞台上的静态文本实例进行交互。也可以在与包含静态文本的 SWF 文件进行交互的 ActionScript 文件中执行类似工作。但是这两种情况下都不能以编程方式对静态文本实例进行实例化。静态文本是在 Flash CS3 创作工具中创建的。

要在 ActionScript 3.0 中创建对现有静态文本的引用，可以遍历显示列表中的项目并分配一个变量。例如：

```
for (var i = 0; i < this.numChildren; i++) {
    var displayitem:DisplayObject = this.getChildAt(i);
    if (displayitem instanceof StaticText) {
        trace("a static text field is item " + i + " on the display list");
     var myFieldLabel:StaticText = StaticText(displayitem);
     trace("and contains the text: " + myFieldLabel.text);
    }
}
```

引用静态文本字段后，您可以在 ActionScript 3.0 中使用该字段的属性。下面的代码附加到时间轴上的一个帧，并假设一个静态文本引用分配有一个名为 `myFieldLabel` 的变量。在示例中，相对于 `myFieldLabel` 的 `x` 和 `y` 值放置名为 `myField` 的动态文本字段，并再次显示 `myFieldLabel` 的值。

```
var myField:TextField = new TextField();
addChild(myField);
myField.x = myFieldLabel.x;
myField.y = myFieldLabel.y + 20;
myField.autoSize = TextFieldAutoSize.LEFT;
myField.text = "and " + myFieldLabel.text
```

## 使用 TextSnapshot 类

如果要以编程方式使用现有静态文本实例，可以使用 flash.text.TextSnapshot 类来与 flash.display.DisplayObjectContainer 的 `textSnapshot` 属性配合工作。也就是说，通过 `DisplayObjectContainer.textSnapshot` 属性创建 TextSnapshot 实例。然后，可以将方法应用于该实例，以检索值或选择部分静态文本。

例如，请在舞台上放置一个包含文本 “TextSnapshot Example”的静态文本字段。将下面的 ActionScript 添加到时间轴中的第 1 帧：

```
var mySnap:TextSnapshot = this.getTextSnapshot();
var count:Number = mySnap.getCount();
mySnap.setSelected(0, 4, true);
mySnap.setSelected(1, 2, false);
var myText:String = mySnap.getSelectedText(false);
trace(myText);
```

TextSnapshot 类对于从所加载的 SWF 中的静态文本字段中获取文本非常有用，这样您就可以在应用程序的其它部分将该文本作为值使用。

# 示例：报纸风格的文本格式设置

“新闻布局”示例设置文本的格式，使文本的外观看起来有点象印刷报纸中的素材。输入文本可以包含标题、副标题和素材正文。在给定显示宽度和高度的情况下，此 “新闻布局”示例将会设置标题和副标题的格式，使其占据整个显示区域的宽度。素材文本将分布在两个或多个列中。

此示例演示以下 ActionScript 编程技巧：

- 扩展 TextField 类
- 加载并应用外部 CSS 文件
- 将 CSS 样式转换为 TextFormat 对象
- 使用 TextLineMetrics 类获取有关文本显示大小的信息

要获取该范例的应用程序文件，请访问 www.adobe.com/go/learn_programmingAS3samples_flash_cn。“新闻布局”应用程序文件位于 Samples/NewsLayout 文件夹中。该应用程序包含以下文件：

| 文件 | 说明 |
|---|---|
| NewsLayout.mxml<br>或<br>NewsLayout.fla | 适用于 Flex (MXML) 或 Flash (FLA) 的应用程序的用户界面。 |
| StoryLayout.as | 排列用于显示的所有新闻素材组件的主要 ActionScript 类。 |

| 文件 | 说明 |
|---|---|
| FormattedTextField.as | 管理本身的 TextFormat 对象的 TextField 类的子类。 |
| HeadlineTextField.as | 调整字体大小以适合需要的宽度的 FormattedTextField 类的子类。 |
| MultiColumnTextField.as | 在两列或多列之间拆分文本的 ActionScript 类。 |
| story.css | 为布局定义文本样式的 CSS 文件。 |
| newsconfig.xml | 包含素材内容的 XML 文件。 |

## 读取外部 CSS 文件

"新闻布局"应用程序开始时读取本地 XML 文件中的素材文本。然后，它读取提供标题、副标题和主体文本的格式设置信息的外部 CSS 文件。

CSS 文件定义三种样式：用于素材的标准段落样式和分别用于标题和副标题的 h1 和 h2 样式。

```
p {
  font-family: Georgia, Times New Roman, Times, _serif;
  font-size: 12;
  leading: 2;
  text-align: justify;
}

h1 {
  font-family: Verdana, Arial, Helvetica, _sans;
  font-size: 20;
  font-weight: bold;
  color: #000099;
  text-align: left;
}

h2 {
  font-family: Verdana, Arial, Helvetica, _sans;
  font-size: 16;
  font-weight: normal;
  text-align: left;
}
```

用于读取外部 CSS 文件的技术与第 451 页的 “加载外部 CSS 文件”中所述的技术相同。加载 CSS 文件后，应用程序执行 `onCSSFileLoaded()` 方法，如下所示。

```
public function onCSSFileLoaded(event:Event):void
{
  this.sheet = new StyleSheet();
  this.sheet.parseCSS(loader.data);

  h1Format = getTextStyle("h1", this.sheet);
  if (h1Format == null)
  {
    h1Format = getDefaultHeadFormat();
  }
  h2Format = getTextStyle("h2", this.sheet);
  if (h2Format == null)
  {
    h2Format = getDefaultHeadFormat();
    h2Format.size = 16;
  }
  displayStory();
}
```

`onCSSFileLoaded()` 方法创建一个新的 StyleSheet 对象并用其分析输入的 CSS 数据。素材的主体文本将显示在 MultiColumnTextField 对象中，该对象可以直接使用 StyleSheet 对象。不过，标题字段使用 HeadlineTextField 类，该类使用 TextFormat 对象进行格式设置。

`onCSSFileLoaded()` 方法调用两次 `getTextStyle()` 方法，将 CSS 样式声明转换为 TextFormat 对象，以便与两个 HeadlineTextField 对象中的每个对象配合使用。
`getTextStyle()` 方法显示如下：

```
public function getTextStyle(styleName:String, ss:StyleSheet):TextFormat
{
  var format:TextFormat = null;

  var style:Object = ss.getStyle(styleName);
  if (style != null)
  {
    var colorStr:String = style.color;
    if (colorStr != null && colorStr.indexOf("#") == 0)
    {
      style.color = colorStr.substr(1);
    }
    format = new TextFormat(style.fontFamily,
                style.fontSize,
                style.color,
                (style.fontWeight == "bold"),
                (style.fontStyle == "italic"),
                (style.textDecoration == "underline"),
                style.url,
                style.target,
```

```
                    style.textAlign,
                    style.marginLeft,
                    style.marginRight,
                    style.textIndent,
                    style.leading);

        if (style.hasOwnProperty("letterSpacing"))
        {
            format.letterSpacing = style.letterSpacing;
        }
    }
    return format;
}
```

CSS 样式声明和 TextFormat 对象的属性名称和属性值的含义不同。`getTextStyle()` 方法可以将 CSS 属性值转换为 TextFormat 对象预期的值。

## 在页面上排列素材元素

StoryLayout 类可以设置标题、副标题和主体文本字段的格式并将这些字段的布局排列为报纸样式。`displayText()` 方法一开始会创建并放置各个字段。

```
public function displayText():void
{
    headlineTxt = new HeadlineTextField(h1Format);
    headlineTxt.wordWrap = true;
    this.addChild(headlineTxt);
    headlineTxt.width = 600;
    headlineTxt.height = 100;
    headlineTxt.fitText(this.headline, 1, true);

    subtitleTxt = new HeadlineTextField(h2Format);
    subtitleTxt.wordWrap = true;
    subtitleTxt.y = headlineTxt.y + headlineTxt.height;
    this.addChild(subtitleTxt);
    subtitleTxt.width = 600;
    subtitleTxt.height = 100;
    subtitleTxt.fitText(this.subtitle, 1, false);

    storyTxt = new MultiColumnTextField(2, 10, 600, 200);
    storyTxt.y = subtitleTxt.y + subtitleTxt.height + 4;
    this.addChild(storyTxt);
    storyTxt.styleSheet = this.sheet;
    storyTxt.htmlText = loremIpsum;
}
```

只要将字段的 `y` 属性设置为等于前一字段的 `y` 属性加上前一字段的高度即可将每个字段放在前一字段的下面。由于 HeadlineTextField 对象和 MultiColumnTextField 对象可以更改其高度以适应内容，因此需要进行这种动态位置计算。

## 更改字体大小以适合字段大小

在给定要显示内容的宽度（以像素为单位）和最多行数的情况下，HeadlineTextField 会更改字体大小以使文本适合字段大小。如果文本很短，则字体大小会很大，从而产生小报样式的标题。如果文本很长，字体大小当然会很小。

下面所示的 `HeadlineTextField.fitText()` 方法的作用就是调整字体大小：

```
public function fitText(msg:String, maxLines:uint = 1, toUpper:Boolean =
  false, targetWidth:Number = -1):uint
{
  this.text = toUpper ? msg.toUpperCase() : msg;

  if (targetWidth == -1)
  {
    targetWidth = this.width;
  }

  var pixelsPerChar:Number = targetWidth / msg.length;

  var pointSize:Number = Math.min(MAX_POINT_SIZE, Math.round(pixelsPerChar *
  1.8 * maxLines));

  if (pointSize < 6)
  {
    // 磅值太小
    return pointSize;
  }

  this.changeSize(pointSize);

  if (this.numLines > maxLines)
  {
    return shrinkText(--pointSize, maxLines);
  }
  else
  {
    return growText(pointSize, maxLines);
  }
}

public function growText(pointSize:Number, maxLines:uint = 1):Number
{
  if (pointSize >= MAX_POINT_SIZE)
  {
    return pointSize;
  }

  this.changeSize(pointSize + 1);
```

```
    if (this.numLines > maxLines)
    {
        // 将它重新设置为以前的大小
        this.changeSize(pointSize);
        return pointSize;
    }
    else
    {
        return growText(pointSize + 1, maxLines);
    }
}

public function shrinkText(pointSize:Number, maxLines:uint=1):Number
{
    if (pointSize <= MIN_POINT_SIZE)
    {
        return pointSize;
    }

    this.changeSize(pointSize);

    if (this.numLines > maxLines)
    {
        return shrinkText(pointSize - 1, maxLines);
    }
    else
    {
        return pointSize;
    }
}
```

HeadlineTextField.fitText() 方法使用简单的递归技术来调整字体大小。首先，该方法推测文本中每个字符的平均像素数，并据此计算起始磅值。然后，它更改文本字段的字体大小并检查文本中的文字是否自动换行，从而产生比最大值更多的文本行。如果文本行过多，则它会调用 shrinkText() 方法减小字体大小并重试。如果文本行不太多，则它会调用 growText() 方法增大字体大小并重试。当字体大小再增加一磅即会产生过多行时，该过程即会停止。

# 在多列之间拆分文本

MultiColumnTextField 类会在多个 TextField 对象中分布文本，然后将这些对象排列成报纸专栏的样式。

`MultiColumnTextField()` 构造函数首先创建一个由 TextField 对象构成的数组，每列一个对象，如下所示：

```
for (var i:int = 0; i < cols; i++)
{
    var field:TextField = new TextField();
    field.autoSize = TextFieldAutoSize.NONE;
    field.wordWrap = true;
    field.styleSheet = this.styleSheet;

    this.fieldArray.push(field);
    this.addChild(field);
}
```

每个 TextField 对象均使用 `addChild()` 方法添加到该数组中，并添加到显示列表中。

只要 StoryLayout 对象的 `text` 属性或 `styleSheet` 属性发生更改，该对象就会调用 `layoutColumns()` 方法来重新显示文本。`layoutColumns()` 方法会调用 `getOptimalHeight()` 方法（如下所示），以计算在给定布局宽度内适合所有文本所需的正确像素高度。

```
public function getOptimalHeight(str:String):int
{
    if (fieldArray.length == 0 || str == "" || str == null)
    {
        return this.preferredHeight;
    }
    else
    {
        var colWidth:int = Math.floor( (this.preferredWidth -
            ((this.numColumns - 1) * gutter)) / this.numColumns);

        var field:TextField = fieldArray[0] as TextField;
        field.width = colWidth;
        field.htmlText = str;

        var linesPerCol:int = Math.ceil(field.numLines / this.numColumns);
        var metrics:TextLineMetrics = field.getLineMetrics(0);
        var prefHeight:int = linesPerCol * metrics.height;
        return prefHeight + 4;
    }
}
```

首先，getOptimalHeight() 方法计算每列的宽度。然后设置数组中第一个 TextField 对象的宽度和 htmlText 属性。getOptimalHeight() 方法使用第一个 TextField 对象计算文本中自动换行文本的总行数，并据此确定每列中应有多少行。接下来，它调用 TextField.getLineMetrics() 方法以检索 TextLineMetrics 对象，该对象包含有关第一行的文本大小的详细信息。TextLineMetrics.height 属性用像素表示文本行的完整高度，包括上升、下降和前导。MultiColumnTextField 对象的最佳高度即为行高度乘以每列行数再加上 4（TextField 对象顶部边框和底部边框各 2 像素）。

以下是完整 layoutColumns() 方法的代码：

```
public function layoutColumns():void
{
  if (this._text == "" || this._text == null)
  {
    return;
  }

  if (this.fitToText)
  {
    this.preferredHeight = this.getOptimalHeight(this._text);
  }

  var colWidth:int = Math.floor( (this.preferredWidth -
    ((numColumns - 1) * gutter)) / numColumns);
  var field:TextField;
  var remainder:String = this._text;
  var fieldText:String = "";

  for (var i:int = 0; i < fieldArray.length; i++)
  {
    field = this.fieldArray[i] as TextField;
    field.width = colWidth;
    field.height = this.preferredHeight;

    field.x = i * (colWidth + gutter);
    field.y = 0;

    field.htmlText = "<p>" + remainder + "</p>";

    remainder = "";
    fieldText = "";

    var linesRemaining:int = field.numLines;
    var linesVisible:int = field.numLines - field.maxScrollV + 1;
    for (var j:int = 0; j < linesRemaining; j++)
    {
      if (j < linesVisible)
      {
        fieldText += field.getLineText(j);
```

```
        }
        else
        {
          remainder +=  field.getLineText(j);
        }
      }

      field.htmlText = "<p>" + fieldText + "</p>";
    }
}
```

通过调用 `getOptimalHeight()` 方法设置 `preferredHeight` 属性后，`layoutColumns()` 方法会循环访问各个 TextField 对象，将每个对象的高度设置为 `preferredHeight` 值。然后，`layoutColumns()` 方法会为每个字段分布刚好足够的文本行，以使每个字段中都不会发生滚动，并且每个连续字段中的文本均会从前一个字段的文本结束处开始。

第 18 章

# 处理位图

18

除了矢量图功能以外，ActionScript 3.0 还包括创建位图图像或操作加载到 SWF 中的外部位图图像的像素数据的能力。使用访问和更改各个像素值的功能，您可以创建自己的滤镜式图像效果并使用内置杂点功能创建纹理和随机杂点。本章将介绍所有这些技术。

## 目录



# 处理位图的基本知识

## 处理位图简介

使用数字图像时，您可能会遇到两种主要的图形类型：位图和矢量。位图图形也称为光栅图形，由排列为矩形网格形式的小方块（像素）组成。矢量图形由以数学方式生成的几何形状（如直线、曲线和多边形）组成。

位图图像用图像的宽度和高度来定义，以像素为量度单位，每个像素包含的位数表示像素包含的颜色数。在使用 RGB 颜色模型的位图图像中，像素由三个字节组成：红、绿和蓝。每个字节包含一个 0 至 255 之间的值。将字节与像素合并时，它们可以产生与艺术混合绘画颜色相似的颜色。例如，一个包含红色字节值 255、绿色字节值 102 和蓝色字节值 0 的像素可以形成明快的橙色。

位图图像的品质由图像分辨率和颜色深度位值共同确定。*分辨率* 与图像中包含的像素数有关。像素数越大，分辨率越高，图像也就越精确。*颜色深度* 与像素可包含的信息量有关。例如，颜色深度值为每像素 16 位的图像无法显示颜色深度为 48 位的图像所具有颜色数。因此， 48 位图像与 16 位图像相比，其阴影具有更高的平滑度。

由于位图图形跟分辨率有关，因此不能很好地进行缩放。当放大位图图像时，这一特性显得尤为突出。通常，放大位图有损其细节和品质。

## 位图文件格式

位图图像可分为几种常见的文件格式。这些格式使用不同类型的压缩算法减小文件大小，并基于图像的最终用途优化图像品质。Adobe Flash Player 支持的位图图像格式有 GIF、JPG 和 PNG。

### GIF

图形交换格式 (GIF) 最初由 CompuServe 于 1987 年开发，作为一种传送 256 色（8 位颜色）图像的方式。此格式提供较小的文件大小，是基于 Web 的图像的理想格式。受此格式的调色板所限，GIF 图像通常不适用于照片，照片通常需要高度的阴影和颜色渐变。GIF 图像允许产生一位透明度，允许将颜色映射为清晰 （或透明）。这可以使网页的背景颜色通过已映射透明度的图像显示出来。

### JPEG

由联合图像专家组 (JPEG) 开发，JPEG（通常写成 JPG）图像格式使用有损压缩算法允许 24 位颜色深度具有很小的文件大小。有损压缩意味着每次保存图像，都会损失图像品质和数据，但会生成更小的文件大小。由于 JPEG 能够显示数百万计的颜色，因此它是照片的理想格式。控制应用于图像的压缩程度的功能使您能够控制图像品质和文件大小。

### PNG

可移植网络图形 (PNG) 格式是作为受专利保护的 GIF 文件格式的开放源替代格式而开发的。 PNG 最多支持 64 位颜色深度，允许使用最多 1600 万种颜色。由于 PNG 是一种比较新的格式，因此一些旧版本浏览器不支持 PNG 文件。与 JPG 不同， PNG 使用无损压缩，这意味着保存图像时不会丢失图像数据。PNG 文件还支持 Alpha 透明度，允许使用最多 256 级透明度。

## 透明位图和不透明位图

使用 GIF 或 PNG 格式的位图图像可以对每个像素添加一个额外字节（Alpha 通道）。此额外像素字节表示像素的透明度值。

GIF 图像允许使用一位透明度，这意味着您可以在 256 色调色板中指定一种透明的颜色。而 PNG 图像最多可以有 256 级透明度。当需要将图像或文本混合到背景中时，此功能特别有用。

ActionScript 3.0 在 BitmapData 类中复制了此额外透明度像素字节。与 PNG 透明度模型类似，`BitmapDataChannel.ALPHA` 常量最多提供 256 级透明度。

# 处理位图的常见任务

以下列表介绍了在 ActionScript 中处理位图图像时您要执行的几项任务：

- 在屏幕上显示位图
- 检索和设置像素颜色值
- 复制位图数据：
  - 创建与位图完全相同的副本
  - 从一个位图的一个颜色通道中向另一个位图的一个颜色通道中复制数据
  - 将屏幕快照显示对象复制到位图中
- 在位图图像中创建杂点和纹理
- 滚动位图

# 重要概念和术语

以下列表包含将在本章中遇到的重要术语：

- Alpha：颜色或图像中的透明度级别（更准确地说是指不透明度）。Alpha 量通常称为 "Alpha 通道" 值。
- ARGB 颜色：一种配色方案，其中每个像素的颜色是红、绿和蓝色值的混合颜色，并将其透明度指定为一个 Alpha 值。
- 颜色通道：通常，将颜色表示为几种基本颜色的混合颜色，对于计算机图形来说，通常是红色、绿色和蓝色。每种基本颜色都视为一个颜色通道；每个颜色通道中的颜色量混合在一起可确定最终颜色。
- 颜色深度：也称为 "位深度"，指专门用于每个像素的计算机内存量，因而可以确定图像中可以显示的可能颜色数。
- 像素：位图图像中的最小信息单位，实际上就是颜色点。
- 分辨率：图像的像素尺寸，它决定图像中包含的精细细节的级别。分辨率通常表示为用像素数表示的宽度和高度。
- RGB 颜色：一种配色方案，其中每个像素的颜色均表示为红、绿和蓝色值的混合颜色。

## 完成本章中的示例

学习本章的过程中，您可能想要测试示例代码。由于本章涉及创建和操作可视内容，因此测试代码包括运行代码以及在创建的 SWF 中查看结果。

要测试本章中的代码示例，请执行以下操作：

1. 创建一个空的 Flash 文档。
2. 在时间轴上选择一个关键帧。
3. 打开 “动作” 面板，将代码复制到 “脚本” 窗格中。
4. 使用 “控制” > “测试影片” 运行程序。

   您将在所创建的 SWF 文件中看到代码的结果。

几乎所有示例代码都包括用于创建位图图像的代码，因此您可以直接测试代码，而无需提供任何位图内容。或者，如果您希望在自己的图像上测试代码，则可以将该图像导入 Adobe Flash CS3 Professional，或将外部图像加载到测试 SWF 并将其位图数据与示例代码一起使用。有关加载外部图像的说明，请参阅第 360 页的 “动态加载显示内容”。

# Bitmap 和 BitmapData 类

处理位图图像的主要 ActionScript 3.0 类是 Bitmap 类 （用于在屏幕上显示位图图像）和 BitmapData 类 （用于访问和操作位图的原始图像数据）。

## 了解 Bitmap 类

作为 DisplayObject 类的子类， Bitmap 类是用于显示位图图像的主要 ActionScript 3.0 类。这些图像可能已经通过 flash.display.Loader 类加载到 Flash 中，或已经使用 `Bitmap()` 构造函数动态创建。从外部源加载图像时，Bitmap 对象只能使用 GIF、JPEG 或 PNG 格式的图像。实例化后，可将 Bitmap 实例视为需要呈现在舞台上的 BitmapData 对象的包装。由于 Bitmap 实例是一个显示对象，因此可以使用显示对象的所有特性和功能来操作 Bitmap 实例。有关使用显示对象的详细信息，请参阅第 319 页的第 12 章 “显示编程”。

## 像素贴紧和平滑

除了所有显示对象常见的功能外， Bitmap 类还提供了特定于位图图像的一些附加功能。

与 Flash 创作工具中的贴紧像素功能类似， Bitmap 类的 `pixelSnapping` 属性可确定 Bitmap 对象是否贴紧最近的像素。此属性接受 PixelSnapping 类中定义的三个常量之一：`ALWAYS`、`AUTO` 和 `NEVER`。

应用像素贴紧的语法为：

```
myBitmap.pixelSnapping = PixelSnapping.ALWAYS;
```

通常，缩放位图图像时，图像会变得模糊或扭曲。若要帮助减少这种扭曲，请使用 BitmapData 类的 smoothing 属性。这是一个布尔值属性，设置为 true 时，缩放图像时，可使图像中的像素平滑或消除锯齿。它可使图像更加清晰、更加自然。

## 了解 BitmapData 类

BitmapData 类位于 flash.display 包中，它可以看作是加载的或动态创建的位图图像中包含的像素的照片快照。此快照用对象中的像素数据的数组表示。BitmapData 类还包含一系列内置方法，可用于创建和处理像素数据。

若要实例化 BitmapData 对象，请使用以下代码：

```
var myBitmap:BitmapData = new BitmapData(width:Number, height:Number,
  transparent:Boolean, fillColor:uinit);
```

width 和 height 参数指定位图的大小；二者的最大值都是 2880 像素。transparent 参数指定位图数据是 (true) 否 (false) 包括 Alpha 通道。fillColor 参数是一个 32 位颜色值，它指定背景颜色和透明度值（如果设置为 true）。以下示例创建一个具有 50% 透明的橙色背景的 BitmapData 对象：

```
var myBitmap:BitmapData = new BitmapData(150, 150, true, 0x80FF3300);
```

若要在屏幕上呈现新创建的 BitmapData 对象，请将此对象分配给或包装到 Bitmap 实例中。为此，可以作为 Bitmap 对象的构造函数的参数形式传递 BitmapData 对象，也可以将此对象分配给现有 Bitmap 实例的 bitmapData 属性。您还必须通过调用将包含该 Bitmap 实例的显示对象容器的 addChild() 或 addChildAt() 方法将该 Bitmap 实例添加到显示列表中。有关使用显示列表的详细信息，请参阅第 329 页的 "在显示列表中添加显示对象"。

以下示例创建一个具有红色填充的 BitmapData 对象，并在 Bitmap 实例中显示此对象：

```
var myBitmapDataObject:BitmapData = new BitmapData(150, 150, false, 0xFF0000);
var myImage:Bitmap = new Bitmap(myBitmapDataObject);
addChild(myImage);
```

# 处理像素

BitmapData 类包含一组用于处理像素数据值的方法。

## 处理单个像素

在像素级别更改位图图像的外观时，您首先需要获取要处理的区域中包含的像素的颜色值。使用 `getPixel()` 方法可读取这些像素值。

`getPixel()` 方法从作为参数传递的一组 x, y （像素）坐标中检索 RGB 值。如果您要处理的像素包括透明度（Alpha 通道）信息，则需要使用 `getPixel32()` 方法。此方法也可以检索 RGB 值，但与 `getPixel()` 不同， `getPixel32()` 返回的值包含表示所选像素的 Alpha 通道 （透明度）值的附加数据。

或者，如果只想更改位图中包含的某个像素的颜色或透明度，则可以使用 `setPixel()` 或 `setPixel32()` 方法。若要设置像素的颜色，只需将 x, y 坐标和颜色值传递到这两种方法之一即可。

以下示例使用 `setPixel()` 在绿色 BitmapData 背景上绘制交叉形状。然后，此示例使用 `getPixel()` 从坐标 50, 50 处的像素中检索颜色值并跟踪返回的值。

```
import flash.display.Bitmap;
import flash.display.BitmapData;

var myBitmapData:BitmapData = new BitmapData(100, 100, false, 0x009900);

for (var i:uint = 0; i < 100; i++)
{
  var red:uint = 0xFF0000;
  myBitmapData.setPixel(50, i, red);
  myBitmapData.setPixel(i, 50, red);
}

var myBitmapImage:Bitmap = new Bitmap(myBitmapData);
addChild(myBitmapImage);

var pixelValue:uint = myBitmapData.getPixel(50, 50);
trace(pixelValue.toString(16));
```

如果要读取一组像素而不是单个像素的值，请使用 `getPixels()` 方法。此方法从作为参数传递的矩形像素数据区域中生成字节数组。字节数组的每个元素（即像素值）都是无符号的整数 （32 位未经相乘的像素值）。

相反，为了更改 （或设置）一组像素值，请使用 `setPixels()` 方法。此方法需要联合使用两个参数（`rect` 和 `inputByteArray`）来输出像素数据 (`inputByteArray`) 的矩形区域 (`rect`)。

从 inputByteArray 中读取（或写入）数据时，会为数组中的每个像素调用 ByteArray.readUnsignedInt() 方法。如果由于某些原因，inputByteArray 未包含像素数据的整个矩形，则该方法会停止处理该点处的图像数据。

必须记住的是，对于获取和设置像素数据，字节数组需要有 32 位 Alpha、红、绿、蓝 (ARGB) 像素值。

以下示例使用 getPixels() 和 setPixels() 方法将一组像素从一个 BitmapData 对象复制到另一个对象：

```
import flash.display.Bitmap;
import flash.display.BitmapData;
import flash.utils.ByteArray;
import flash.geom.Rectangle;

var bitmapDataObject1:BitmapData = new BitmapData(100, 100, false,
  0x006666FF);
var bitmapDataObject2:BitmapData = new BitmapData(100, 100, false,
  0x00FF0000);

var rect:Rectangle = new Rectangle(0, 0, 100, 100);
var bytes:ByteArray = bitmapDataObject1.getPixels(rect);

bytes.position = 0;
bitmapDataObject2.setPixels(rect, bytes);

var bitmapImage1:Bitmap = new Bitmap(bitmapDataObject1);
addChild(bitmapImage1);
var bitmapImage2:Bitmap = new Bitmap(bitmapDataObject2);
addChild(bitmapImage2);
bitmapImage2.x = 110;
```

## 像素级别冲突检测

BitmapData.hitTest() 方法可以在位图数据和另一个对象或点之间执行像素级别冲突检测。

BitmapData.hitTest() 方法接受五个参数：

- firstPoint (Point)：此参数指在其上执行点击测试的第一个 BitmapData 的左上角的像素位置。
- firstAlphaThreshold (uint)：此参数指定对于此点击测试视为不透明的最高 Alpha 通道值。
- secondObject (Object)：此参数表示影响区域。secondObject 对象可以是 Rectangle、Point、Bitmap 或 BitmapData 对象。此对象表示在其上执行冲突检测的点击区域。
- secondBitmapDataPoint (Point)：此可选参数用于在第二个 BitmapData 对象中定义像素位置。只有当 secondObject 的值为 BitmapData 对象时，才使用此参数。默认值为 null。

- `secondAlphaThreshold` (uint): 此可选参数表示在第二个 BitmapData 对象中视为不透明的最高 Alpha 通道值。默认值为 1。只有当 `secondObject` 是一个 BitmapData 对象且两个 BitmapData 对象都透明时，才使用此参数。

在不透明图像上执行冲突检测时，请牢记， ActionScript 会将图像视为完全不透明的矩形（或边框）。或者，在透明的图像上执行像素级别点击测试时，需要两个图像都是透明的。除此之外， ActionScript 还使用 Alpha 阈值参数来确定像素在哪点开始从透明变为不透明。

以下示例创建三个位图图像并使用两个不同冲突点 （一个返回 false，另一个返回 true）检查像素冲突：

```
import flash.display.Bitmap;
import flash.display.BitmapData;
import flash.geom.Point;

var bmd1:BitmapData = new BitmapData(100, 100, false, 0x000000FF);
var bmd2:BitmapData = new BitmapData(20, 20, false, 0x00FF3300);

var bm1:Bitmap = new Bitmap(bmd1);
this.addChild(bm1);

// Create a red square.
var redSquare1:Bitmap = new Bitmap(bmd2);
this.addChild(redSquare1);
redSquare1.x = 0;

// Create a second red square.
var redSquare2:Bitmap = new Bitmap(bmd2);
this.addChild(redSquare2);
redSquare2.x = 150;
redSquare2.y = 150;

// Define the point at the top-left corner of the bitmap.
var pt1:Point = new Point(0, 0);
// Define the point at the center of redSquare1.
var pt2:Point = new Point(20, 20);
// Define the point at the center of redSquare2.
var pt3:Point = new Point(160, 160);

trace(bmd1.hitTest(pt1, 0xFF, pt2)); // true
trace(bmd1.hitTest(pt1, 0xFF, pt3)); // false
```

# 复制位图数据

若要从一个图像向另一个图像中复制位图数据，可以使用多种方法: clone()、copyPixels()、copyChannel() 和 draw()。

正如名称的含义一样，clone() 方法允许您将位图数据从一个 BitmapData 对象克隆或采样到另一个对象。调用此方法时，此方法返回一个新的 BitmapData 对象，它是与被复制的原始实例完全一样的克隆。

以下示例克隆橙色 （父级）正方形的一个副本，并将克隆放在原始父级正方形的旁边:

```
import flash.display.Bitmap;
import flash.display.BitmapData;

var myParentSquareBitmap:BitmapData = new BitmapData(100, 100, false,
  0x00ff3300);
var myClonedChild:BitmapData = myParentSquareBitmap.clone();

var myParentSquareContainer:Bitmap = new Bitmap(myParentSquareBitmap);
this.addChild(myParentSquareContainer);

var myClonedChildContainer:Bitmap = new Bitmap(myClonedChild);
this.addChild(myClonedChildContainer);
myClonedChildContainer.x = 110;
```

copyPixels() 方法是一种从一个 BitmapData 对象向另一个对象复制像素的快速简便的方法。该方法会拍摄源图像的矩形快照 （由 sourceRect 参数定义），并将其复制到另一个矩形区域 （大小相等）。新 “粘贴”的矩形位置在 destPoint 参数中定义。

copyChannel() 方法从源 BitmapData 对象中采集预定义的颜色通道值 （Alpha、红、绿或蓝），并将此值复制到目标 BitmapData 对象的通道中。调用此方法不会影响目标 BitmapData 对象中的其它通道。

draw() 方法将源 sprite、影片剪辑或其它显示对象中的图形内容绘制或呈现在新位图上。使用 matrix、 colorTransform、 blendMode 和目标 clipRect 参数，可以修改新位图的呈现方式。此方法使用 Flash Player 矢量渲染器生成数据。

调用 draw() 时，需要将源对象（sprite、影片剪辑或其它显示对象）作为第一个参数传递，如下所示:

```
myBitmap.draw(movieClip);
```

如果源对象在最初加载后应用了变形（颜色、矩阵等等），则不能将这些变形复制到新对象。如果想要将变形复制到新位图，则需要将 transform 属性的值从原始对象复制到使用新 BitmapData 对象的 Bitmap 对象的 transform 属性中。

# 使用杂点功能制作纹理

若要修改位图的外观，可以使用 noise() 方法或 perlinNoise() 方法对位图应用杂点效果。可以把杂点效果比作未调谐的电视屏幕的静态外观。

若要对位图应用杂点效果，请使用 noise() 方法。此方法对位图图像的指定区域中的像素应用随机颜色值。

此方法接受五个参数：

- randomSeed (int)：决定图案的随机种子数。不管名称具有什么样的含义，只要传递的数字相同，此数字就会生成相同的结果。为了获得真正的随机结果，请使用 Math.random() 方法为此参数传递随机数字。
- low (uint)：此参数指要为每个像素生成的最低值 （0 至 255）。默认值为 0。将此参数设置为较低值会产生较暗的杂点图案，而将此参数设置为较高值会产生较亮的图案。
- high (uint)：此参数指要为每个像素生成的最高值 （0 至 255）。默认值为 255。将此参数设置为较低值会产生较暗的杂点图案，而将此参数设置为较高值会产生较亮的图案。
- channelOptions (uint)：此参数指定将向位图对象的哪个颜色通道应用杂点图案。此数字可以是四个颜色通道 ARGB 值的任意组合。默认值是 7。
- grayScale (Boolean)：设置为 true 时，此参数对位图像素应用 randomSeed 值，可有效地褪去图像中的所有颜色。此参数不影响 Alpha 通道。默认值为 false。

以下示例创建一个位图图像，并对它应用蓝色杂点图案：

```
import flash.display.Bitmap;
import flash.display.BitmapData;

var myBitmap:BitmapData = new BitmapData(250, 250,false, 0xff000000);
myBitmap.noise(500, 0, 255, BitmapDataChannel.BLUE,false);
var image:Bitmap = new Bitmap(myBitmap);
addChild(image);
```

如果想要创建更好的有机外观纹理，请使用 perlinNoise() 方法。perlinNoise() 方法可生成逼真、有机的纹理，是用于烟雾、云彩、水、火或爆炸的理想图案。

由于 perlinNoise() 方法是由算法生成的，因此它使用的内存比基于位图的纹理少。但还是会对处理器的使用有影响，特别是对于旧计算机，会降低 Flash 内容的处理速度，使屏幕重新绘制的速度比帧频慢。这主要是因为需要进行浮点计算，以便处理 Perlin 杂点算法。

此方法接受九个参数 （前六个是必需参数）：

- baseX (Number)：决定创建的图案的 x （大小）值。
- baseY (Number)：决定创建的图案的 y （大小）值。
- numOctaves (uint)：要组合以创建此杂点的 octave 函数或各个杂点函数的数目。octave 数目越大，创建的图像越精细，但这需要更多的处理时间。

- randomSeed (int): 随机种子数的功能与在 noise() 函数中的功能完全相同。为了获得真正的随机结果，请使用 Math.random() 方法为此参数传递随机数字。
- stitch (Boolean)：如果设置为 true，则此方法尝试缝合（或平滑）图像的过渡边缘以形成无缝的纹理，用于作为位图填充进行平铺。
- fractalNoise (Boolean)：此参数与此方法生成的渐变的边缘有关。如果设置为 true，则此方法生成的碎片杂点会对效果的边缘进行平滑处理。如果设置为 false，则将生成湍流。带有湍流的图像具有可见的不连续性渐变，可以使用它处理更接近锐化的视觉效果，例如，火焰或海浪。
- channelOptions (uint)：channelOptions 参数的功能与在 noise() 方法中的功能完全相同。它指定对哪个颜色通道（在位图上）应用杂点图案。此数字可以是四个颜色通道 ARGB 值的任意组合。默认值是 7。
- grayScale (Boolean): grayScale 参数的功能与在 noise() 方法中的功能完全相同。如果设置为 true，则对位图像素应用 randomSeed 值，可有效地褪去图像中的所有颜色。默认值为 false。
- offsets (Array)：对应于每个 octave 的 x 和 y 偏移的点数组。通过处理偏移值，可以平滑滚动图像层。偏移数组中的每个点将影响一个特定的 octave 杂点函数。默认值为 null。

以下示例创建一个 150 x 150 像素的 BitmapData 对象，该对象调用 perlinNoise() 方法来生成绿色和蓝色的云彩效果：

```
import flash.display.Bitmap;
import flash.display.BitmapData;

var myBitmapDataObject:BitmapData = new BitmapData(150, 150, false,
  0x00FF0000);

var seed:Number = Math.floor(Math.random() * 100);
var channels:uint = BitmapDataChannel.GREEN | BitmapDataChannel.BLUE
myBitmapDataObject.perlinNoise(100, 80, 6, seed, false, true, channels,
  false, null);

var myBitmap:Bitmap = new Bitmap(myBitmapDataObject);
addChild(myBitmap);
```

# 滚动位图

设想您创建了一个街道图应用程序，每次用户移动该图时，都需要您更新视图（即使该图只移动了几个像素）。

创建此功能的一种方式是，每次用户移动街道图时，均重新呈现包含更新的街道图视图的新图像。或者，创建一个大型图像，并使用 `scroll()` 方法。

`scroll()` 方法可以复制屏幕上的位图，然后将它粘贴到由 (x, y) 参数指定的新偏移位置。如果位图的一部分恰巧在舞台以外，则会产生图像发生移位的效果。与计时器函数（或 `enterFrame` 事件）配合使用时，可以使图像呈现动画或滚动效果。

以下示例采用前面的 Perlin 杂点示例并生成较大的位图图像（其四分之三呈现在舞台外面）。然后应用 `scroll()` 方法和一个 `enterFrame` 事件侦听器，使图像在对角线向下方向偏移一个像素。每次输入帧时均会调用此方法，因此，随着图像向下滚动，图像位于屏幕以外的部分会呈现在舞台上。

```
import flash.display.Bitmap;
import flash.display.BitmapData;

var myBitmapDataObject:BitmapData = new BitmapData(1000, 1000, false,
  0x00FF0000);
var seed:Number = Math.floor(Math.random() * 100);
var channels:uint = BitmapDataChannel.GREEN | BitmapDataChannel.BLUE;
myBitmapDataObject.perlinNoise(100, 80, 6, seed, false, true, channels,
  false, null);

var myBitmap:Bitmap = new Bitmap(myBitmapDataObject);
myBitmap.x = -750;
myBitmap.y = -750;
addChild(myBitmap);

addEventListener(Event.ENTER_FRAME, scrollBitmap);

function scrollBitmap(event:Event):void
{
  myBitmapDataObject.scroll(1, 1);
}
```

# 示例：动画处理使用屏幕外位图的 sprite

许多 Flash 游戏都是在屏幕上一次显示数百个图像动画。本位图动画示例在一个大的屏幕外位图上绘制数百个小位图或 sprite，然后将这个位图写在屏幕上，从而大大加快动画速度。有关此示例和下载源代码的说明，请访问 www.adobe.com/go/learn_fl_bitmaps_cn。

第 19 章

# 处理视频

19

Flash 视频是 Internet 上的一项优秀技术。然而，视频的传统演示形式（在下方具有一个进度栏和一些控制按钮的矩形屏幕中）只是视频在 Flash 应用程序中的一种可能使用方式。通过 ActionScript，您可以微调和控制视频的加载、演示和回放。

## 目录

# 视频基础知识

## 视频处理简介

Adobe Flash Player 的一个重要功能是可以使用 ActionScript，以操作其它可视内容（如图像、动画、文本等）的方式显示和操作视频信息。

在 Adobe Flash CS3 Professional 中创建 Flash 视频 (FLV) 文件时，您可以选择视频的外观，包括常用的回放控件。不过，您不一定要局限于可用的选项。使用 ActionScript 可以精确控制视频的加载、显示和回放，这意味着您可以创建自己的视频播放器外观，也可以按照所需的任何非传统方式使用视频。

在 ActionScript 中使用视频涉及多个类的联合使用：

- Video 类：舞台上的实际视频内容框是 Video 类的一个实例。 Video 类是一种显示对象，因此可以使用适用于其它显示对象的同样的技术（比如定位、应用变形、应用滤镜和混合模式等）进行操作。
- NetStream 类：在加载将由 ActionScript 控制的视频文件时，将使用一个 NetStream 实例来表示该视频内容的源，在本例中为视频数据流。使用 NetStream 实例也涉及 NetConnection 对象的使用，该对象是到视频文件的连接，它好比是视频数据馈送的通道。
- Camera 类：在通过连接到用户计算机的摄像头处理视频数据时，会使用一个 Camera 实例来表示视频内容的源，即用户的摄像头和它所提供的视频数据。

在加载外部视频时，您可以从标准 Web 服务器加载文件以便进行渐进式下载回放，也可以使用由专门的服务器（如 Adobe 的 Macromedia® Flash® Media Server）传送的视频流。

## 常见视频任务

本章介绍了以下您将希望执行的与视频相关的任务：

- 显示和控制屏幕上的视频
- 加载外部 FLV 文件
- 处理视频文件中的元数据和提示点信息
- 捕获和显示从用户摄像头输入的视频

## 重要概念和术语

- 提示点：一个可以放在视频文件内特定时刻的标记，例如，可用作书签以便定位到该时刻或提供与该时刻相关联的其它数据。
- 编码：以一种格式接收视频数据并将其转换为另一种视频数据格式的过程；例如，接收高分辨率的源视频并将其转换为适合于 Internet 传送的格式。
- 帧：单一的一段视频信息；每一帧都类似于一个代表某一时刻的快照的静止图像。通过按顺序高速播放各个帧，可产生动画视觉效果。
- 关键帧：包含帧的完整信息的视频帧。关键帧后面的其它帧仅包含有关它们与关键帧之间的差异的信息，而不包含完整的帧信息。
- 元数据：有关视频文件的信息，可嵌入在视频文件中并可在加载视频时检索。
- 渐进式下载：当视频文件从标准 Web 服务器传送时，会使用渐进式下载来加载视频数据，这意味着会按顺序加载视频信息。其好处是不必等待整个文件下载完毕即可开始播放视频；不过，它会阻止您向前跳到视频中尚未加载的部分。
- 流式传输：渐进式下载的一种替代方法，使用流式传输 （有时称为 “实流”）技术和一台专用视频服务器通过 Internet 传送视频。使用流式传输，用于查看视频的计算机不必一次下载整个视频。为了加快下载速度，在任何时刻，计算机均只需要整个视频信息的一部分。由于使用一台专用服务器来控制视频内容的传送，因此可以在任何时刻访问视频的任何部分，而无需等待其下载完毕后才能进行访问。

## 完成本章中的示例

学习本章的过程中，您可能想要自己动手测试一些示例代码清单。由于本章是有关在 ActionScript 中使用视频的，因此，本章中的许多代码清单都涉及处理视频对象 （可能是在 Flash 创作工具中创建并放置在舞台上的对象，也可能是使用 ActionScript 创建的对象）。测试范例将涉及在 Flash Player 中查看结果，以了解代码对视频的影响。

多数示例代码清单都操作 Video 对象而不显式创建该对象。要测试本章中的这些代码清单，请执行以下操作：

1. 创建一个空的 Flash 文档。
2. 在时间轴上选择一个关键帧。
3. 打开 “动作”面板，将代码清单复制到 “脚本”窗格中。
4. 如有必要，请打开 “库”面板。
5. 从 “库”面板菜单中，选择 “新建视频”。
6. 在 “视频属性”对话框中，输入新视频元件的名称，然后在 “类型”字段中选择 “视频 ( 受 ActionScript 控制 )”。单击 “确定”创建一个视频元件。
7. 将视频元件的一个实例从 “库”面板拖动到舞台上。

8. 使视频实例保持选中状态，在 “属性” 检查器中，为其指定实例名称。名称应与示例代码清单中视频实例使用的名称相匹配，例如，如果代码清单操作名为 `vid` 的 Video 对象，则应将舞台实例也命名为 `vid`。
9. 使用 “控制” > “测试影片” 运行程序。

   在屏幕上，您将看到按照代码清单所指定的要求操作视频的结果。

本章中的一些示例代码清单除了包括示例代码以外，还包括类定义。在这些列表中，除了前几步之外，在测试 SWF 之前，还需要创建用在示例中的类。要创建在示例代码清单中定义的类，请执行以下操作：

1. 请确保已经保存了将用于测试的 FLA 文件。
2. 从主菜单中选择 “文件” > “新建”。
3. 在 “新建文档” 对话框的 “类型” 部分，选择 “ActionScript 文件”。单击 “确定” 创建新的 ActionScript 文件。
4. 将类定义代码从示例复制到 ActionScript 文档中。
5. 从主菜单中选择 “文件” > “保存”。将该文件保存在 Flash 文档所在的目录中。文件名应与代码清单中的类的名称一致。例如，如果代码清单定义一个名为 “VideoTest” 的类，则将 ActionScript 文件保存为 “VideoTest.as”。
6. 返回到 Flash 文档。
7. 使用 “控制” > “测试影片” 运行程序。

   您将在屏幕上看到示例的结果。

测试示例代码清单的其它技术在第 53 页的 “测试本章内的示例代码清单”中有更详细的介绍。

# 了解 Flash 视频 (FLV) 格式

FLV 文件格式包含用 Flash Player 编码以便于传送的音频和视频数据。例如，如果您有 QuickTime 或 Windows Media 视频文件，便可使用编码器（如 Flash Video Encoder 或 Sorenson™ Squeeze）将该文件转换为 FLV 文件。

可以通过将视频导入到 Flash 创作工具，然后导出为 FLV 文件来创建 FLV 文件。可以使用 “FLV 导出” 插件从受支持的视频编辑应用程序中导出 FLV 文件。

使用外部 FLV 文件可以提供使用导入的视频时不可用的某些功能：

- 无需降低回放速度就可以在 Flash 文档中使用较长的视频剪辑。可以使用缓存内存的方式来播放外部 FLV 文件，这意味着可以将大型文件分成若干个小片段存储，对其进行动态访问，这种方式比嵌入的视频文件所需的内存更少。

- 外部 FLV 文件可以和它所在的 Flash 文档具有不同的帧速率。例如，可以将 Flash 文档帧速率设置为 30 帧 / 秒 (fps)，并将视频帧速率设置为 21 fps。与嵌入的视频相比，此项设置可使您更好地控制视频，确保视频顺畅地回放。此项设置还允许您在不改变现有 Flash 内容的前提下以不同的帧速率播放 FLV 文件。
- 利用外部 FLV 文件，Flash 文档回放就不必在视频文件进行加载时中断。导入的视频文件有时可能需要中断文档回放来执行某些功能，例如，访问 CD-ROM 驱动器。FLV 文件可以独立于 Flash 文档执行功能，因此不会中断回放。
- 对于外部 FLV 文件，为视频内容加字幕更加简单，这是因为您可以使用事件处理函数访问视频的元数据。

**提示**

若要从 Web 服务器加载 FLV 文件，则可能需要向您的 Web 服务器注册文件扩展名和 MIME 类型；请查看您的 Web 服务器文档。FLV 文件的 MIME 类型是 `video/x-flv`。有关详细信息，请参阅第 504 页的 "关于配置 FLV 文件以便在服务器上托管"。

# 了解 Video 类

使用 Video 类可以直接在应用程序中显示实时视频流，而无需将其嵌入 SWF 文件中。可以使用 `Camera.getCamera()` 方法捕获并播放实时视频。还可以使用 Video 类通过 HTTP 或在本地文件系统中回放 FLV 文件。在项目中使用 Video 有多种不同方法：

- 使用 NetConnection 和 NetStream 类动态加载 FLV 并在 Video 对象中显示视频。
- 从用户摄像头捕获输入。
- 使用 FLVPlayback 组件。



舞台上 Video 对象的实例是 Video 类的实例。

尽管 Video 类位于 flash.media 包中，但它继承自 flash.display.DisplayObject 类，因此，所有显示对象功能（如矩阵转换和滤镜）也适用于 Video 实例。

有关详细信息，请参阅第 339 页的 "处理显示对象"、第 371 页的 "处理几何结构" 和第 403 页的 "过滤显示对象"。

# 加载视频文件

使用 NetStream 和 NetConnection 类加载视频是一个多步骤过程：

1. 第一步是创建一个 NetConnection 对象。如果连接到没有使用服务器（如 Adobe 的 Flash Media Server 2 或 Adobe Flex）的本地 FLV 文件，则使用 NetConnection 类可通过向 `connect()` 方法传递值 `null`，来从 HTTP 地址或本地驱动器播放流式 FLV 文件。

```
var nc:NetConnection = new NetConnection();
nc.connect(null);
```

2. 第二步是创建一个 NetStream 对象（该对象将 NetConnection 对象作为参数）并指定要加载的 FLV 文件。以下代码片断将 NetStream 对象连接到指定的 NetConnection 实例，并加载 SWF 文件所在的目录中名为 video.flv 的 FLV：

```
var ns:NetStream = new NetStream(nc);
ns.addEventListener(AsyncErrorEvent.ASYNC_ERROR, asyncErrorHandler);
ns.play("video.flv");
function asyncErrorHandler(event:AsyncErrorEvent):void
{
  // 忽略错误
}
```

3. 第三步是创建一个新的 Video 对象，并使用 Video 类的 `attachNetStream()` 方法附加以前创建的 NetStream 对象。然后可以使用 `addChild()` 方法将该视频对象添加到显示列表中，如以下代码片断所示：

```
var vid:Video = new Video();
vid.attachNetStream(ns);
addChild(vid);
```

输入上面的代码后，Flash Player 将尝试加载 SWF 文件所在目录中的 video.flv 视频文件。

**提示** 若要从 Web 服务器加载 FLV 文件，则可能需要向您的 Web 服务器注册文件扩展名和 MIME 类型；请查看您的 Web 服务器文档。 FLV 文件的 MIME 类型是 `video/x-flv`。有关详细信息，请参阅第 504 页的 “关于配置 FLV 文件以便在服务器上托管”。

# 控制视频回放

NetStream 类提供了四个用于控制视频回放的主要方法：

pause()：暂停视频流的回放。如果视频已经暂停，则调用此方法将不会执行任何操作。

resume()：恢复回放暂停的视频流。如果视频已在播放，则调用此方法将不会执行任何操作。

seek()：搜寻最接近指定位置（从流的开始位置算起的偏移量，以秒为单位）的关键帧。

togglePause()：暂停或恢复流的回放。



没有 stop() 方法。为了停止视频流，必须暂停回放并找到视频流的开始位置。



play() 方法不会恢复回放，它用于加载视频文件。

以下示例演示如何使用多个不同的按钮控制视频。若要运行下面的示例，请创建一个新文档，并在工作区中添加 4 个按钮实例（pauseBtn、playBtn、stopBtn 和 togglePauseBtn）：

```
var nc:NetConnection = new NetConnection();
nc.connect(null);

var ns:NetStream = new NetStream(nc);
ns.addEventListener(AsyncErrorEvent.ASYNC_ERROR, asyncErrorHandler);
ns.play("video.flv");
function asyncErrorHandler(event:AsyncErrorEvent):void
{
  // 忽略错误
}

var vid:Video = new Video();
vid.attachNetStream(ns);
addChild(vid);

pauseBtn.addEventListener(MouseEvent.CLICK, pauseHandler);
playBtn.addEventListener(MouseEvent.CLICK, playHandler);
stopBtn.addEventListener(MouseEvent.CLICK, stopHandler);
togglePauseBtn.addEventListener(MouseEvent.CLICK, togglePauseHandler);

function pauseHandler(event:MouseEvent):void
{
  ns.pause();
}
function playHandler(event:MouseEvent):void
{
  ns.resume();
}
function stopHandler(event:MouseEvent):void
{
```

```
    // 暂停流并将播放头移回到
    // 流的开始位置。
    ns.pause();
    ns.seek(0);
}
function togglePauseHandler(event:MouseEvent):void
{
    ns.togglePause();
}
```

播放视频的同时单击 `pauseBtn` 按钮实例会导致视频文件暂停。如果视频已经暂停，则单击此按钮不会执行任何操作。如果之前暂停了回放，则单击 `playBtn` 按钮实例会恢复视频回放；如果视频已在播放，则单击该按钮不会执行任何操作。

## 检测视频流的末尾

为了侦听视频流的开始和末尾，需要向 NetStream 实例添加一个事件侦听器以侦听 `netStatus` 事件。以下代码演示如何在视频回放过程中侦听不同代码：

```
ns.addEventListener(NetStatusEvent.NET_STATUS, statusHandler);
function statusHandler(event:NetStatusEvent):void
{
    trace(event.info.code)
}
```

上面这段代码的输出如下：

```
NetStream.Play.Start
NetStream.Buffer.Empty
NetStream.Buffer.Full
NetStream.Buffer.Empty
NetStream.Buffer.Full
NetStream.Buffer.Empty
NetStream.Buffer.Full
NetStream.Buffer.Flush
NetStream.Play.Stop
NetStream.Buffer.Empty
NetStream.Buffer.Flush
```

您专门想要侦听的两段代码为“NetStream.Play.Start”和“NetStream.Play.Stop”，它们会在视频回放到达开始和末尾时发出信号。下面的代码片断使用 switch 语句来过滤这两段代码并输出一条消息：

```
function statusHandler(event:NetStatusEvent):void
{
    switch (event.info.code)
    {
        case "NetStream.Play.Start":
            trace("Start [" + ns.time.toFixed(3) + " seconds]");
            break;
```

```
        case "NetStream.Play.Stop":
            trace("Stop [" + ns.time.toFixed(3) + " seconds]");
            break;
    }
}
```

通过侦听 netStatus 事件 (NetStatusEvent.NET_STATUS)，您可以生成一个视频播放器，它在当前视频完成播放后加载播放列表中的下一个视频。

# 流式传输视频文件

若要流式传输 Flash Media Server 中的文件，可以使用 NetConnection 和 NetStream 类连接到远程服务器实例并播放指定的流。要指定实时消息传递协议 (RTMP) 服务器，请向 NetConnection.connect() 方法传递所需的 RTMP URL（例如 “rtmp://localhost/appName/appInstance”），而不传递 null。若要播放指定 Flash Media Server 中的指定实时流或录制流，请为 NetStream.play() 方法传递一个由 NetStream.publish() 发布的实时数据的标识名称，或一个要回放的录制文件名称。有关详细信息，请参阅 Flash Media Server 文档。

# 了解提示点

并非所有 FLV 文件都包含提示点。虽然有在现有 FLV 文件中嵌入提示点的工具，但提示点通常是在 FLV 编码过程中嵌入在 FLV 文件中的。

您可以对 Flash 视频使用几种不同类型的提示点。可以使用 ActionScript 与在创建 FLV 文件时嵌入到 FLV 文件中的提示点进行交互，也可以与用 ActionScript 创建的提示点进行交互。

- 导航提示点：您可以在编码 FLV 文件时，将导航提示点嵌入到 FLV 流和 FLV 元数据包中。使用导航提示点可以使用户搜索到文件的指定部分。
- 事件提示点：您可以在编码 FLV 文件时，将事件提示点嵌入到 FLV 流和 FLV 元数据包中。还可以编写代码来处理在 FLV 回放期间于指定点上触发的事件。
- ActionScript 提示点：使用 ActionScript 代码创建的外部提示点。您可以编写代码来触发这些与视频回放有关的提示点。这些提示点的精确度要低于嵌入的提示点（最高时相差 1/10 秒），因为视频播放器单独跟踪这些提示点。

由于导航提示点会在指定的提示点位置创建一个关键帧，因此可以使用代码将视频播放器的播放头移动到该位置。您可以在 FLV 文件中设置一些特殊点，让用户搜索这些点。例如，视频可能会具有多个章节或段，在这种情况下您就可以在视频文件中嵌入导航提示点，以此方式来控制视频。

如果您计划创建一个应用程序，希望用户能在其中导航至提示点，则应在编码文件时创建并嵌入提示点，而不应使用 ActionScript 提示点。您应将提示点嵌入到 FLV 文件中，因为这些提示点需要更加精确的处理。有关在 FLV 文件中嵌入提示点的详细信息，请参阅《使用 Flash》中的"嵌入提示点"。

您可以通过编写 ActionScript 来访问提示点参数。提示点参数是从 `onCuePoint` 回调处理函数接收的事件对象的一部分。

若要在视频到达特定提示点时在代码中触发特定动作，请使用 `NetStream.onCuePoint` 事件处理函数。有关详细信息，请参阅第 486 页的"为 onCuePoint 和 onMetaData 编写回调方法"。

# 为 onCuePoint 和 onMetaData 编写回调方法

当播放器到达特定提示点或收到特定元数据时，您可以在应用程序中触发动作。若要触发此类动作，请使用 `onCuePoint` 和 `onMetaData` 事件处理函数。必须为这些处理函数编写回调方法，否则，Flash Player 可能会引发错误。例如，以下代码播放 SWF 文档所在文件夹中名为 video.flv 的 FLV 文件：

```
var nc:NetConnection = new NetConnection();
nc.connect(null);

var ns:NetStream = new NetStream(nc);
ns.addEventListener(AsyncErrorEvent.ASYNC_ERROR, asyncErrorHandler);
ns.play("video.flv");
function asyncErrorHandler(event:AsyncErrorEvent):void
{
  trace(event.text);
}

var vid:Video = new Video();
vid.attachNetStream(ns);
addChild(vid);
```

上面的代码加载一个名为 video.flv 的本地 FLV 文件并侦听要调度的 `asyncError` (`AsyncErrorEvent.ASYNC_ERROR`)。当本机异步代码中引发异常时调度此事件。在本例中，当 FLV 中包含元数据或提示点信息，并且未定义相应的侦听器时调度此事件。如果您对视频文件的元数据或提示点信息不感兴趣，则可以使用上面的代码处理 `asyncError` 事件并忽略错误。如果 FLV 中具有元数据或多个提示点，则会输出以下信息：

```
错误 #2095: flash.net.NetStream 无法调用回调 onMetaData。
错误 #2095: flash.net.NetStream 无法调用回调 onCuePoint。
错误 #2095: flash.net.NetStream 无法调用回调 onCuePoint。
错误 #2095: flash.net.NetStream 无法调用回调 onCuePoint。
```

发生错误的原因是 NetStream 对象找不到 onMetaData 或 onCuePoint 回调方法。在应用程序中定义这些回调方法有多种方式：

- 将 NetStream 对象的 client 属性设置为一个 Object
- 创建自定义类并定义用于处理回调方法的方法
- 扩展 NetStream 类并添加处理回调方法的方法
- 扩展 NetStream 类并使其为动态类
- 将 NetStream 对象的 client 属性设置为 this

## 将 NetStream 对象的 client 属性设置为一个 Object

通过将 client 属性设置为一个 Object 或设置为 NetStream 的一个子类，可以重新发送 onMetaData 和 onCuePoint 回调方法或彻底忽略这些方法。以下示例演示如何使用空的 Object 忽略这些回调方法而不侦听 asyncError 事件：

```
var nc:NetConnection = new NetConnection();
nc.connect(null);

var customClient:Object = new Object();

var ns:NetStream = new NetStream(nc);
ns.client = customClient;
ns.play("video.flv");

var vid:Video = new Video();
vid.attachNetStream(ns);
addChild(vid);
```

如果想要侦听 onMetaData 或 onCuePoint 回调方法，则需要定义用于处理这些回调方法的方法，如以下代码片断所示：

```
var customClient:Object = new Object();
customClient.onMetaData = metaDataHandler;
function metaDataHandler(infoObject:Object):void
{
  trace("metadata");
}
```

上面的代码侦听 onMetaData 回调方法并调用 metaDataHandler() 方法，后者会输出一个字符串。如果 Flash Player 遇到一个提示点，那么即使未定义 onCuePoint 回调方法，也不会生成错误。

## 创建自定义类并定义用于处理回调方法的方法

以下代码将 NetStream 对象的 client 属性设置为一个自定义类 CustomClient，该类为回调方法定义处理函数：

```
var nc:NetConnection = new NetConnection();
nc.connect(null);

var ns:NetStream = new NetStream(nc);
ns.client = new CustomClient();
ns.play("video.flv");

var vid:Video = new Video();
vid.attachNetStream(ns);
addChild(vid);
```

CustomClient 类如下所示：

```
package
{
  public class CustomClient
  {
    public function onMetaData(infoObject:Object):void
    {
      trace("metadata");
    }
  }
}
```

CustomClient 类为 onMetaData 回调处理函数定义一个处理函数。如果遇到一个提示点并调用了 onCuePoint 回调处理函数，则会调度一个 asyncError 事件 (AsyncErrorEvent.ASYNC_ERROR)，显示 "flash.net.NetStream 无法调用回调 onCuePoint"。为了防止发生此错误，需要在 CustomClient 类中定义一个 onCuePoint 回调方法，或者为 asyncError 事件定义一个事件处理函数。

## 扩展 NetStream 类并添加处理回调方法的方法

以下代码创建 CustomNetStream 类的一个实例，CustomNetStream 类在后面的代码清单中定义：

```
var ns:CustomNetStream = new CustomNetStream();
ns.play("video.flv");

var vid:Video = new Video();
vid.attachNetStream(ns);
addChild(vid);
```

以下代码清单定义 CustomNetStream 类，该类扩展 NetStream 类、处理必要的 NetConnection 对象的创建并处理 `onMetaData` 和 `onCuePoint` 回调处理函数方法：

```
package
{
  import flash.net.NetConnection;
  import flash.net.NetStream;
  public class CustomNetStream extends NetStream
  {
    private var nc:NetConnection;
    public function CustomNetStream()
    {
      nc = new NetConnection();
      nc.connect(null);
      super(nc);
    }
    public function onMetaData(infoObject:Object):void
    {
      trace("metadata");
    }
    public function onCuePoint(infoObject:Object):void
    {
      trace("cue point");
    }
  }
}
```

如果您想要重命名 CustomNetStream 类中的 `onMetaData()` 和 `onCuePoint()` 方法，可以使用以下代码：

```
package
{
  import flash.net.NetConnection;
  import flash.net.NetStream;
  public class CustomNetStream extends NetStream
  {
    private var nc:NetConnection;
    public var onMetaData:Function;
    public var onCuePoint:Function;
    public function CustomNetStream()
    {
      onMetaData = metaDataHandler;
      onCuePoint = cuePointHandler;
      nc = new NetConnection();
      nc.connect(null);
      super(nc);
    }
    private function metaDataHandler(infoObject:Object):void
    {
      trace("metadata");
    }
```

```
    private function cuePointHandler(infoObject:Object):void
    {
      trace("cue point");
    }
  }
}
```

## 扩展 NetStream 类并使其为动态类

可以扩展 NetStream 类并使其子类为动态类，以便可以动态添加 onCuePoint 和 onMetaData 回调处理函数。以下代码清单演示了这一过程：

```
var ns:DynamicCustomNetStream = new DynamicCustomNetStream();
ns.play("video.flv");

var vid:Video = new Video();
vid.attachNetStream(ns);
addChild(vid);
```

DynamicCustomNetStream 类如下所示：

```
package
{
  import flash.net.NetConnection;
  import flash.net.NetStream;
  public dynamic class DynamicCustomNetStream extends NetStream
  {
    private var nc:NetConnection;
    public function DynamicCustomNetStream()
    {
      nc = new NetConnection();
      nc.connect(null);
      super(nc);
    }
  }
}
```

由于 DynamicCustomNetStream 类为动态类，因此，即使 onMetaData 和 onCuePoint 回调处理函数没有处理函数，也不会引发错误。如果想要为 onMetaData 和 onCuePoint 回调处理函数定义方法，可以使用以下代码：

```
var ns:DynamicCustomNetStream = new DynamicCustomNetStream();
ns.onMetaData = metaDataHandler;
ns.onCuePoint = cuePointHandler;
ns.play("http://www.helpexamples.com/flash/video/cuepoints.flv");

var vid:Video = new Video();
vid.attachNetStream(ns);
addChild(vid);

function metaDataHandler(infoObject:Object):void
```

```
{
  trace("metadata");
}
function cuePointHandler(infoObject:Object):void
{
  trace("cue point");
}
```

## 将 NetStream 对象的 client 属性设置为 this

通过将 `client` 属性设置为 `this`， Flash Player 会在当前范围内查找 `onMetaData()` 和 `onCuePoint()` 方法。以下示例演示了这一效果：

```
var nc:NetConnection = new NetConnection();
nc.connect(null);

var ns:NetStream = new NetStream(nc);
ns.client = this;
ns.play("video.flv");

var vid:Video = new Video();
vid.attachNetStream(ns);
addChild(vid);
```

调用 `onMetaData` 或 `onCuePoint` 回调处理函数时，如果不存在处理该回调的方法时，不会生成错误。若要处理这些回调处理函数，请在代码中创建 `onMetaData()` 和 `onCuePoint()` 方法，如以下代码片断所示：

```
function onMetaData(infoObject:Object):void
{
  trace("metadata");
}
function onCuePoint(infoObject:Object):void
{
  trace("cue point");
}
```

# 使用提示点

以下示例使用一个简单的 for..in 循环来遍历 onCuePoint 回调处理函数的 infoObject 参数中的每个属性，并在收到提示点数据时输出一条消息：

```
var nc:NetConnection = new NetConnection();
nc.connect(null);

var ns:NetStream = new NetStream(nc);
ns.client = this;
ns.play("video.flv");

var vid:Video = new Video();
vid.attachNetStream(ns);
addChild(vid);

function onCuePoint(infoObject:Object):void
{
  var key:String;
  for (key in infoObject)
  {
    trace(key + ": " + infoObject[key]);
  }
}
```

显示以下输出：

```
parameters:
name: point1
time: 0.418
type: navigation
```

此代码使用多种技术之一设置在其上调用该回调方法的对象。您可以使用其它技术；有关详细信息，请参阅为 onCuePoint 和 onMetaData 编写回调方法。

# 使用视频元数据

您可以使用 onMetaData 回调处理函数来查看 FLV 文件中的元数据信息。元数据包含 FLV 文件的相关信息，如持续时间、宽度、高度和帧速率。添加到 FLV 文件中的元数据信息取决于编码 FLV 文件时所使用的软件或添加元数据信息时所使用的软件。

```
var nc:NetConnection = new NetConnection();
nc.connect(null);

var ns:NetStream = new NetStream(nc);
ns.client = this;
ns.play("video.flv");

var vid:Video = new Video();
```

```
vid.attachNetStream(ns);
addChild(vid);

function onMetaData(infoObject:Object):void
{
  var key:String;
  for (key in infoObject)
  {
    trace(key + ": " + infoObject[key]);
  }
}
```

如果 FLV 文件中包含提示点和音频，则上面的代码生成类似于以下内容的代码：

```
width: 320
audiodelay: 0.038
canSeekToEnd: true
height: 213
cuePoints: ,,
audiodatarate: 96
duration: 16.334
videodatarate: 400
framerate: 15
videocodecid: 4
audiocodecid: 2
```

| 提示 | 如果您的视频没有音频，则与音频相关的元数据信息（如 audiodatarate）将返回 undefined，因为在编码期间没有将音频信息添加到元数据中。 |
|---|---|

在上面的代码中，未显示提示点信息。为了显示提示点元数据，可以使用以下函数，该函数会以递归方式显示 Object 中的项目：

```
function traceObject(obj:Object, indent:uint = 0):void
{
  var indentString:String = "";
  var i:uint;
  var prop:String;
  var val:*;
  for (i = 0; i < indent; i++)
  {
    indentString += "\t";
  }
  for (prop in obj)
  {
    val = obj[prop];
    if (typeof(val) == "object")
    {
      trace(indentString + " " + j + ": [Object]");
      traceObject(val, indent + 1);
    }
    else
```

```
      {
        trace(indentString + " " + prop + ": " + val);
      }
    }
  }
```

使用上面的代码片断跟踪 onMetaData() 方法中的 infoObject 参数会输出以下结果:

```
width: 320
audiodatarate: 96
audiocodecid: 2
videocodecid: 4
videodatarate: 400
canSeekToEnd: true
duration: 16.334
audiodelay: 0.038
height: 213
framerate: 15
cuePoints: [Object]
  0: [Object]
    parameters: [Object]
      lights: beginning
    name: point1
    time: 0.418
    type: navigation
  1: [Object]
    parameters: [Object]
      lights: middle
    name: point2
    time: 7.748
    type: navigation
  2: [Object]
    parameters: [Object]
      lights: end
    name: point3
    time: 16.02
    type: navigation
```

# onMetaData 的信息对象

下表显示视频元数据的可能值

| 参数 | 描述 |
|---|---|
| audiocodecid | 一个数字，指示已使用的音频编解码器 （编码 / 解码技术）。 |
| audiodatarate | 一个数字，指示音频的编码速率，以每秒千字节为单位。 |
| audiodelay | 一个数字，指示原始 FLV 文件的 “time 0”在 FLV 文件中保持多长时间。为了正确同步音频，视频内容需要有少量的延迟。 |
| canSeekToEnd | 一个布尔值，如果 FLV 文件是用最后一帧 （它允许定位到渐进式下载影片剪辑的末尾）上的关键帧编码的，则该值为 `true`。如果 FLV 文件不是用最后一帧上的关键帧编码的，则该值为 `false`。 |
| cuePoints | 嵌入在 FLV 文件中的提示点对象组成的数组，每个提示点对应一个对象。如果 FLV 文件不包含任何提示点，则值是未定义的。每个对象都具有以下属性：<br>■ `type`：一个字符串，它将提示点的类型指定为 “navigation”或 “event”。<br>■ `name`：一个字符串，表示提示点的名称。<br>■ `time`：一个数字，表示以秒为单位的提示点时间，精确到小数点后三位 （毫秒）。<br>■ `parameters`：一个可选对象，具有用户在创建提示点时指定的名称 - 值对。 |
| duration | 一个数字，以秒为单位指定 FLV 文件的持续时间。 |
| framerate | 一个数字，表示 FLV 文件的帧速率。 |
| height | 一个数字，以像素为单位表示 FLV 文件的高度。 |
| videocodecid | 一个数字，表示用于对视频进行编码的编解码器版本。 |
| videodatarate | 一个数字，表示 FLV 文件的视频数据速率。 |
| width | 一个数字，以像素为单位表示 FLV 文件的宽度。 |

下表显示 `videocodecid` 参数的可能值：

| videocodecid | 编解码器名称 |
|---|---|
| 2 | Sorenson H.263 |
| 3 | 屏幕视频 （仅限 SWF 7 和更高版本） |
| 4 | VP6 （仅限 SWF 8 和更高版本） |
| 5 | 带有 Alpha 通道的 VP6 视频 （仅限 SWF 8 和更高版本） |

下表显示 `audiocodecid` 参数的可能值：

| audiocodecid | 编解码器名称 |
|---|---|
| 0 | 未压缩 |
| 1 | ADPCM |
| 2 | mp3 |
| 5 | Nellymoser 8kHz 单声 |
| 6 | Nellymoser |

# 捕获摄像头输入

除了外部视频文件外，附加到用户计算机上的摄像头也可以作为您使用 ActionScript 进行显示和操作的视频数据的来源。Camera 类是 ActionScript 中内置的机制，用于使用计算机摄像头。

## 了解 Camera 类

使用 Camera 对象可以连接到用户的本地摄像头并在本地广播视频（回放给用户），或将其广播到远程服务器 （比如 Flash Media Server）。

使用 Camera 类可以访问有关用户摄像头的以下各种信息：

- Flash Player 可以使用用户计算机上安装的哪些摄像头
- 是否安装了摄像头
- 是否允许 Flash Player 访问用户摄像头
- 哪个摄像头当前处于活动状态
- 正在捕获的视频的宽度和高度

Camera 类包括多个有用的方法和属性，通过这些方法和属性可以使用 Camera 对象。例如，静态的 `Camera.names` 属性包含当前安装在用户计算机上的摄像头的名称数组。您也可以使用 `name` 属性显示当前处于活动状态的摄像头的名称。

## 在屏幕上显示摄像头内容

连接到摄像头所需的代码比使用 NetConnection 和 NetStream 类加载 FLV 的代码少。由于需要有用户许可才能让 Flash Player 连接到摄像头，并且只有在连接到摄像头后才能访问摄像头，因此， Camera 类的使用可能很快就会变得非常麻烦。

以下代码演示如何使用 Camera 类连接到用户的本地摄像头：

```
var cam:Camera = Camera.getCamera();
var vid:Video = new Video();
vid.attachCamera(cam);
addChild(vid);
```




Camera 类不具有构造函数方法。若要创建新的 Camera 实例，请使用静态 `Camera.getCamera()` 方法。

## 设计摄像头应用程序

在编写需要连接到用户摄像头的应用程序时，需要在代码中考虑以下事项：

- 检查用户当前是否安装了摄像头。
- 检查用户是否显式允许 Flash Player 访问其摄像头。出于安全原因，播放器会显示 "Flash Player 设置"对话框，让用户选择允许还是拒绝对其摄像头的访问。这样可以防止 Flash Player 在未经用户许可的情况下连接到其摄像头并广播视频流。如果用户单击允许，则应用程序即可连接到用户的摄像头。如果用户单击拒绝，则应用程序将无法访问用户的摄像头。应用程序始终应适当地处理这两种情况。

## 连接到用户摄像头

连接到用户摄像头时，执行的第一步是通过创建一个类型为 Camera 的变量并将其初始化为静态 `Camera.getCamera()` 方法的返回值来创建一个新的 Camera 实例。

下一步是创建一个新的视频对象并向其附加 Camera 对象。

第三步是向显示列表中添加该视频对象。由于 Camera 类不会扩展 DisplayObject 类，它不能直接添加到显示列表中，因此需要执行第 2 步和第 3 步。若要显示摄像头捕获的视频，需要创建一个新的视频对象并调用 `attachCamera()` 方法。

以下代码演示这三个步骤:

```
var cam:Camera = Camera.getCamera();
var vid:Video = new Video();
vid.attachCamera(cam);
addChild(vid);
```

注意，如果用户未安装摄像头， Flash Player 将不显示任何内容。

在实际情况下，您需要对应用程序执行其它步骤。有关详细信息，请参阅验证是否已安装摄像头和检测摄像头的访问权限。

## 验证是否已安装摄像头

在尝试对 Camera 实例使用任何方法或属性之前，您需要验证用户是否已安装了摄像头。检查用户是否已安装摄像头有两种方式:

- 检查静态 `Camera.names` 属性，该属性包含可用摄像头名称的数组。此数组通常具有一个或几个字符串，因为多数用户不太可能同时安装多个摄像头。以下代码演示如何检查 `Camera.names` 属性以查看用户是否具有可用的摄像头:

  ```
  if (Camera.names.length > 0)
  {
    trace("用户未安装摄像头。");
  }
  else
  {
    var cam:Camera = Camera.getCamera(); // 获取默认摄像头。
  }
  ```

- 检查静态 `Camera.getCamera()` 方法的返回值。如果没有摄像头可用或未安装摄像头，则此方法将返回 `null`，否则返回对 Camera 对象的引用。以下代码演示如何检查 `Camera.getCamera()` 方法以查看用户是否具有可用的摄像头:

  ```
  var cam:Camera = Camera.getCamera();
  if (cam == null)
  {
    trace("用户未安装摄像头。");
  }
  else
  {
    trace("用户至少安装了 1 个摄像头。");
  }
  ```

由于 Camera 类不会扩展 DisplayObject 类，因此不能通过使用 `addChild()` 方法将它直接添加到显示列表中。为了显示摄像头捕获的视频，您需要创建一个新的 Video 对象并对 Video 实例调用 `attachCamera()` 方法。

以下代码片断演示在存在摄像头的情况下如何附加摄像头；如果不存在摄像头，Flash Player 将不显示任何内容：

```
var cam:Camera = Camera.getCamera();
if (cam != null)
{
  var vid:Video = new Video();
  vid.attachCamera(cam);
  addChild(vid);
}
```

## 检测摄像头的访问权限

在可以显示摄像头输出之前，用户必须显式允许 Flash Player 访问该摄像头。在调用 attachCamera() 方法后， Flash Player 会显示 "Flash Player 设置"对话框，提示用户允许或拒绝 Flash Player 访问摄像头或麦克风。如果用户单击 "允许"按钮，则会在舞台上的 Video 实例中显示摄像头输出。如果用户单击 "拒绝"按钮，则 Flash Player 将无法连接到摄像头，且 Video 对象将不显示任何内容。

如果用户未安装摄像头，Flash Player 将不会显示任何内容。如果用户安装了摄像头，Flash Player 将会显示"Flash Player 设置"对话框，提示用户允许或拒绝 Flash Player 访问摄像头。如果用户允许访问其摄像头，则会向用户显示视频，否则不会显示任何内容。

如果想要检测用户是否允许访问其摄像头，可以侦听摄像头的 status 事件 (StatusEvent.STATUS)，如以下代码所示：

```
var cam:Camera = Camera.getCamera();
if (cam != null)
{
  cam.addEventListener(StatusEvent.STATUS, statusHandler);
  var vid:Video = new Video();
  vid.attachCamera(cam);
  addChild(vid);
}
function statusHandler(event:StatusEvent):void
{
  // 当用户在 "Flash Player 设置"对话框中单击
  //  "允许"或 "拒绝"按钮时调度此事件。
  trace(event.code); // "Camera.Muted"或 "Camera.Unmuted"
}
```

一旦用户单击 “允许” 或 “拒绝” 后，即会调用 statusHandler() 函数。使用以下两种方法之一可以检测用户单击了哪个按钮：

- statusHandler() 函数的 event 参数包含一个 code 属性，其中包含字符串 “Camera.Muted” 或 “Camera.Unmuted”。如果值为 “Camera.Muted”，则说明用户单击了 “拒绝” 按钮， Flash Player 将无法访问该摄像头。在下面的代码片段中您会看到此情况的一个示例：

```
function statusHandler(event:StatusEvent):void
{
  switch (event.code)
  {
    case "Camera.Muted":
      trace("用户单击了 “拒绝”。");
      break;
    case "Camera.Unmuted":
      trace("用户单击了 “接受”。");
      break;
  }
}
```

- Camera 类包含一个名为 muted 的只读属性，它可以指明用户在 Flash Player 的“隐私”面板中是拒绝访问摄像头 (true) 还是允许访问摄像头 (false)。在下面的代码片段中您会看到此情况的一个示例：

```
function statusHandler(event:StatusEvent):void
{
  if (cam.muted)
  {
    trace("用户单击了 “拒绝”。");
  }
  else
  {
    trace("用户单击了 “接受”。");
  }
}
```

通过检查将要调度的 status 事件，您可以编写处理用户接受或拒绝访问摄像头的代码并进行相应的清理。例如，如果用户单击 “拒绝” 按钮，则您可以向用户显示一条消息，说明他们如果想要参加视频聊天的话，需要单击 “允许”；或者，您也可以确保删除显示列表中的 Video 对象以释放系统资源。

## 最优化视频品质

默认情况下，Video 类的新实例为 320 像素宽乘以 240 像素高。为了最优化视频品质，应始终确保视频对象与 Camera 对象返回的视频具有相同的尺寸。使用 Camera 类的 `width` 和 `height` 属性，您可以获取 Camera 对象的宽度和高度，然后将该视频对象的 `width` 和 `height` 属性设置为与 Camera 对象的尺寸相符，也可以将 Camera 对象的宽度和高度传递给 Video 类的构造函数方法，如以下代码片断所示：

```
var cam:Camera = Camera.getCamera();
if (cam != null)
{
    var vid:Video = new Video(cam.width, cam.height);
    vid.attachCamera(cam);
    addChild(vid);
}
```

由于 `getCamera()` 方法返回对 Camera 对象的引用（在没有可用摄像头时返回 `null`），因此，即使用户拒绝访问其摄像头，您也可以访问 Camera 对象的方法和属性。这样可以使用摄像头的本机高度和宽度设置视频实例的尺寸。

```
var vid:Video;
var cam:Camera = Camera.getCamera();

if (cam == null)
{
    trace("找不到可用的摄像头。");
}
else
{
    trace("找到摄像头：" + cam.name);
    cam.addEventListener(StatusEvent.STATUS, statusHandler);
    vid = new Video();
    vid.attachCamera(cam);
}
function statusHandler(event:StatusEvent):void
{
    if (cam.muted)
    {
        trace("无法连接到活动摄像头。");
    }
    else
    {
        // 调整 Video 对象的大小，使之与摄像头设置相符，并
        // 将该视频添加到显示列表中。
        vid.width = cam.width;
        vid.height = cam.height;
        addChild(vid);
    }
    // 删除 status 事件侦听器。
    cam.removeEventListener(StatusEvent.STATUS, statusHandler);
}
```

## 监视回放条件

Camera 类包含多个属性，这些属性允许您监视 Camera 对象的当前状态。例如，以下代码使用一个 Timer 对象和一个文本字段实例在显示列表上显示摄像头的若干属性：

```
var vid:Video;
var cam:Camera = Camera.getCamera();
var tf:TextField = new TextField();
tf.x = 300;
tf.autoSize = TextFieldAutoSize.LEFT;
addChild(tf);

if (cam != null)
{
  cam.addEventListener(StatusEvent.STATUS, statusHandler);
  vid = new Video();
  vid.attachCamera(cam);
}
function statusHandler(event:StatusEvent):void
{
  if (!cam.muted)
  {
    vid.width = cam.width;
    vid.height = cam.height;
    addChild(vid);
    t.start();
  }
  cam.removeEventListener(StatusEvent.STATUS, statusHandler);
}

var t:Timer = new Timer(100);
t.addEventListener(TimerEvent.TIMER, timerHandler);
function timerHandler(event:TimerEvent):void
{
  tf.text = "";
  tf.appendText("activityLevel: " + cam.activityLevel + "\n");
  tf.appendText("bandwidth: " + cam.bandwidth + "\n");
  tf.appendText("currentFPS: " + cam.currentFPS + "\n");
  tf.appendText("fps: " + cam.fps + "\n");
  tf.appendText("keyFrameInterval: " + cam.keyFrameInterval + "\n");
  tf.appendText("loopback: " + cam.loopback + "\n");
  tf.appendText("motionLevel: " + cam.motionLevel + "\n");
  tf.appendText("motionTimeout: " + cam.motionTimeout + "\n");
  tf.appendText("quality: " + cam.quality + "\n");
}
```

每 1/10 秒（100 毫秒），即会调度一次 Timer 对象的 `timer` 事件，并且 `timerHandler()` 函数会更新显示列表上的文本字段。

## 向服务器发送视频

如果您要生成涉及 Video 或 Camera 对象的更为复杂的应用程序，可以使用 Flash Media Server 提供的流媒体功能和开发环境组合来创建媒体应用程序并将它提供给广泛的目标用户。开发人员可以使用这一组合来创建应用程序，如 Video on Demand、实时 Web 事件广播和 mp3 流，以及视频博客、视频消息传送和多媒体聊天环境。有关详细信息，请参阅 Flash Media Server 在线文档，地址为 http://livedocs.macromedia.com/fms/2/docs/。

# 高级主题

以下主题介绍使用视频的一些特殊问题。

## Flash Player 与编码的 FLV 文件的兼容性

Flash Player 7 支持用 Sorenson™ Spark™ 视频编解码器编码的 FLV 文件。 Flash Player 8 支持用 Flash Professional 8 中的 Sorenson Spark 或 On2 VP6 编码器编码的 FLV 文件。On2 VP6 视频编解码器支持 Alpha 通道。不同的 Flash Player 版本支持 FLV 的方式也不同。有关详细信息，请参阅下表：

| 编解码器 | SWF 文件版本（发布版本） | Flash Player 版本（回放所需要的版本） |
| --- | --- | --- |
| Sorenson Spark | 6 | 6、7 或 8 |
| | 7 | 7, 8 |
| On2 VP6 | 6 | 8* |
| | 7 | 8 |
| | 8 | 8 |

* 如果 SWF 文件加载 FLV 文件，则只要用户使用 Flash Player 8 查看 SWF 文件，您就可以使用 On2 VP6 视频，而不必为 Flash Player 8 重新发布 SWF 文件。只有 Flash Player 8 既支持发布又支持回放 On2 VP6 视频。

## 关于配置 FLV 文件以便在服务器上托管

在处理 FLV 文件时，您可能需要配置服务器以便于处理 FLV 文件格式。多用途 Internet 邮件扩展 (MIME) 是标准的数据规范，允许您通过 Internet 连接发送非 ASCII 文件。 Web 浏览器和电子邮件客户端经过配置，可以解释多种 MIME 类型，因此它们可以发送和接收视频、音频、图形和格式化文本。若要从 Web 服务器加载 FLV 文件，则可能需要向您的 Web 服务器注册文件扩展名和 MIME 类型，因此应检查您的 Web 服务器文档。FLV 文件的 MIME 类型是 `video/x-flv`。下面列出了 FLV 文件类型的完整信息：

- Mime 类型：video/x-flv
- 文件扩展名：.flv
- 必需参数：无
- 可选参数：无
- 编码注意事项: FLV 文件是二进制文件；有些应用程序可能需要设置应用程序 / 八位字节流子类型。
- 安全问题：无
- 已发布的规范：www.adobe.com/go/flashfileformat_cn。

Microsoft 更改了在 Microsoft Internet 信息服务 (IIS) 6.0 Web 服务器中处理流媒体的方式，不再采用早期版本中的处理方式。早期版本的 IIS 不需要对 Flash 视频流做任何修改。在 Windows 2003 附带的默认 Web 服务器 IIS 6.0 中，服务器需要借助 MIME 类型来确认 FLV 文件为流媒体。

当采用流式媒体的方式加载外部 FLV 文件的 SWF 文件被置于 Microsoft Windows Server® 2003 上，并在浏览器中查看时，可以正确播放 SWF 文件，但 FLV 视频却不能采用流式媒体的方式加载。这个问题会影响到放置在 Windows Server 2003 服务器上的所有 FLV 文件，包括用早期版本的 Flash 创作工具（Adobe 的 Macromedia Flash Video Kit for Dreamweaver MX 2004）制作的那些文件。如果在其它操作系统上对这些文件进行测试，则这些文件可以正常工作。

有关配置 Microsoft Windows 2003 和 Microsoft IIS Server 6.0 以采用流式媒体的方式加载 FLV 视频的信息，请访问 www.adobe.com/go/tn_19439_cn。

## 关于在 Macintosh 上将本地 FLV 文件设定为目标

如果您尝试从 Apple® Macintosh® 计算机的非系统驱动器上用以斜杠 (/) 表示的相对路径来播放本地 FLV，将无法播放视频。非系统驱动器包括（但不限于） CD-ROM、分区的硬盘、可移除的存储媒体以及连接的存储设备。



注意：导致失败发生的原因是操作系统的局限，而非 Flash Player 的问题。

若要从 Macintosh 的非系统驱动器上播放 FLV 文件，需使用基于冒号记号 (:) 的绝对路径来代替基于斜杠记号 (/) 的路径引用该文件。下面的列表显示了这两种记号的不同：

- 基于斜杠的记号：myDrive/myFolder/myFLV.flv
- 基于冒号的记号：(Mac OS®) myDrive:myFolder:myFLV.flv

您还可以为想让 Macintosh 回放的 CD-ROM 创建一个放映文件。有关 Mac OS CD-ROM 和 FLV 文件的最新信息，请访问 www.adobe.com/go/3121b301_cn。

# 示例：视频自动唱片点唱机

以下示例生成一个简单的视频自动唱片点唱机，它可以动态加载一组视频并按顺序回放。您可以生成一个可以让用户浏览一系列视频教程的应用程序，也可以指定在传送用户请求的视频之前应回放的广告。此示例演示了 ActionScript 3.0 的下列功能：

- 根据视频文件的回放进度更新播放头
- 侦听并分析视频文件的元数据
- 处理网络流中的特定代码
- 加载、播放、暂停和停止动态加载的 FLV
- 根据网络流的元数据调整显示列表上视频对象的大小

要获取该范例的应用程序文件，请访问
www.adobe.com/go/learn_programmingAS3samples_flash_cn。"视频自动唱片点唱机"应用程序文件位于 Samples/VideoJukebox 文件夹中。该应用程序包含以下文件：

| 文件 | 描述 |
| --- | --- |
| VideoJukebox.as | 此类提供了应用程序的主要功能。 |
| VideoJukebox.fla | 适用于 Flash 的主应用程序文件。 |
| playlist.xml | 列出要向视频自动唱片点唱机中加载哪些视频文件的文件。 |

## 加载外部视频播放列表文件

外部 playlist.xml 文件指定要加载的视频以及回放这些文件的顺序。为了加载该 XML 文件，您需要使用一个 URLLoader 对象和一个 URLRequest 对象，如以下代码所示：

```
uldr = new URLLoader();
uldr.addEventListener(Event.COMPLETE, xmlCompleteHandler);
uldr.load(new URLRequest(PLAYLIST_XML_URL));
```

此代码放置在 VideoJukebox 类的构造函数内，因此，在运行其它任何代码之前，会首先加载该文件。一旦 XML 文件加载完成后，即会调用 `xmlCompleteHandler()` 方法，该方法会将该外部文件分析为一个 XML 对象，如以下代码所示：

```
private function xmlCompleteHandler(event:Event):void
{
  playlist = XML(event.target.data);
  videosXML = playlist.video;
  main();
}
```

播放列表 XML 对象包含来自该外部文件的原始 XML，而 videosXML 是一个 XMLList 对象，它仅包含视频节点。以下代码片断显示了一个示例 playlist.xml 文件：

```
<videos>
  <video url="video/caption_video.flv" />
  <video url="video/cuepoints.flv" />
  <video url="video/water.flv" />
</videos>
```

最后，`xmlCompleteHandler()` 方法将调用 `main()` 方法，被调用的方法会在显示列表上设置各个组件实例以及用于加载外部 FLV 文件的 NetConnection 和 NetStream 对象。

## 创建用户界面

若要构建用户界面，您需要将 5 个 Button 实例拖到显示列表上并为其指定以下实例名称：`playButton`、`pauseButton`、`stopButton`、`backButton` 和 `forwardButton`。

对于这些 Button 实例中的每个实例，您需要为 `click` 事件分配一个处理函数，如以下代码片断所示：

```
playButton.addEventListener(MouseEvent.CLICK, buttonClickHandler);
pauseButton.addEventListener(MouseEvent.CLICK, buttonClickHandler);
stopButton.addEventListener(MouseEvent.CLICK, buttonClickHandler);
backButton.addEventListener(MouseEvent.CLICK, buttonClickHandler);
forwardButton.addEventListener(MouseEvent.CLICK, buttonClickHandler);
```

`buttonClickHandler()` 方法使用 switch 语句来确定单击了哪个按钮实例，如以下代码所示：

```
private function buttonClickHandler(event:MouseEvent):void
{
  switch (event.currentTarget)
  {
    case playButton:
      ns.resume();
      break;
    case pauseButton:
      ns.togglePause();
      break;
    case stopButton:
      ns.pause();
      ns.seek(0);
      break;
    case backButton:
      playPreviousVideo();
      break;
    case forwardButton:
      playNextVideo();
      break;
  }
}
```

接下来，向显示列表中添加一个 Slider 实例，并为其指定实例名称 `volumeSlider`。以下代码将该滑块实例的 `liveDragging` 属性设置为 `true`，并为该滑块实例的 `change` 事件定义一个事件侦听器：

```
volumeSlider.value = volumeTransform.volume;
volumeSlider.minimum = 0;
volumeSlider.maximum = 1;
volumeSlider.snapInterval = 0.1;
volumeSlider.tickInterval = volumeSlider.snapInterval;
volumeSlider.liveDragging = true;
volumeSlider.addEventListener(SliderEvent.CHANGE, volumeChangeHandler);
```

向显示列表中添加一个 ProgressBar 实例，并为其指定实例名称 `positionBar`。将其 `mode` 属性设置为 manual，如以下代码片断所示：

```
positionBar.mode = ProgressBarMode.MANUAL;
```

最后，向显示列表中添加一个 Label 实例，并为其指定实例名称 `positionLabel`。此 Label 实例的值将由计时器实例设置

## 侦听视频对象的元数据

当 Flash Player 遇到已加载的每个视频的元数据时，会对 NetStream 对象的 `client` 属性调用 `onMetaData()` 回调处理函数。以下代码初始化一个对象并设置指定的回调处理函数：

```
client = new Object();
client.onMetaData = metadataHandler;
```

`metadataHandler()` 方法会将其数据复制到在代码前面部分中定义的 meta 属性中。这可使您在应用程序运行过程中随时访问当前视频的元数据。接下来，调整舞台上视频对象的大小，使其与从元数据中返回的尺寸相符。最后，根据当前正在播放的视频的尺寸移动并调整 positionBar 进度栏实例的大小。以下代码包含完整的 `metadataHandler()` 方法：

```
private function metadataHandler(metadataObj:Object):void
{
  meta = metadataObj;
  vid.width = meta.width;
  vid.height = meta.height;
  positionBar.move(vid.x, vid.y + vid.height);
  positionBar.width = vid.width;
}
```

## 动态加载 Flash 视频

为了动态加载每个 Flash 视频，应用程序应使用 NetConnection 和 NetStream 对象。以下代码创建一个 NetConnection 对象并将 `null` 传递给 `connect()` 方法。通过指定 `null`，Flash Player 会连接到本地服务器上的视频，而不连接到服务器（如 Flash Media Server）。

以下代码创建 NetConnection 和 NetStream 实例，为 `netStatus` 事件定义事件侦听器，并将 `client` 对象分配给 `client` 属性：

```
nc = new NetConnection();
nc.connect(null);

ns = new NetStream(nc);
ns.addEventListener(NetStatusEvent.NET_STATUS, netStatusHandler);
ns.client = client;
```

每当视频的状态发生改变时即会调用 `netStatusHandler()` 方法。这包括视频开始或停止回放时、缓冲时或找不到视频流时。以下代码列出了 `netStatusHandler()` 事件：

```
private function netStatusHandler(event:NetStatusEvent):void
{
  try
  {
    switch (event.info.code)
    {
      case "NetStream.Play.Start":
        t.start();
        break;
```

```
        case "NetStream.Play.StreamNotFound":
        case "NetStream.Play.Stop":
          t.stop();
          playNextVideo();
          break;
      }
    }
    catch (error:TypeError)
    {
      // 忽略任何错误。
    }
}
```

上面的代码计算 info 对象的 code 属性，并过滤出为 “NetStream.Play.Start”、“NetStream.Play.StreamNotFound” 或 “NetStream.Play.Stop” 的代码。其它所有代码均被忽略。如果网络流已开始，代码会启动用于更新播放头的 Timer 实例。如果找不到网络流或网络流已停止，则会停止 Timer 实例，应用程序将尝试播放播放列表中的下一个视频。

每次执行 Timer 时，`positionBar` 进度栏实例都会通过调用 ProgressBar 类的 `setProgress()` 方法来更新其当前位置，并且会用当前视频已经过的时间和总时间来更新 `positionLabel` Label 实例。

```
private function timerHandler(event:TimerEvent):void
{
  try
  {
    positionBar.setProgress(ns.time, meta.duration);
    positionLabel.text = ns.time.toFixed(1) + " of "
  meta.duration.toFixed(1) + " seconds";
  }
  catch (error:Error)
  {
    // 忽略此错误。
  }
}
```

## 控制视频音量

通过在 NetStream 对象上设置 soundTransform 属性，您可以控制动态加载视频的音量。视频自动唱片点唱机应用程序允许您通过更改 volumeSlider Slider 实例的值来修改音量级别。以下代码演示如何通过将 Slider 组件的值赋给 SoundTransform 对象（在 NetStream 对象上设置为 soundTransform 属性）来更改音量级别：

```
private function volumeChangeHandler(event:SliderEvent):void
{
    volumeTransform.volume = event.value;
    ns.soundTransform = volumeTransform;
}
```

## 控制视频回放

应用程序的其余部分是在视频到达视频流末尾或用户跳到上一个或下一个视频时控制视频的回放。

下面的方法可以从当前所选索引的 XMLList 中检索视频 URL：

```
private function getVideo():String
{
    return videosXML[idx].@url;
}
```

playVideo() 方法会调用 NetStream 对象上的 play() 方法以加载当前所选的视频：

```
private function playVideo():void
{
    var url:String = getVideo();
    ns.play(url);
}
```

playPreviousVideo() 方法使当前视频索引递减、调用 playVideo() 方法来加载新的视频文件并将进度栏设置为可见：

```
private function playPreviousVideo():void
{
    if (idx > 0)
    {
        idx--;
        playVideo();
        positionBar.visible = true;
    }
}
```

最后一个方法 playNextVideo() 使视频索引递增并调用 playVideo() 方法。如果当前视频是播放列表中的最后一个视频，则会对 Video 对象调用 clear() 方法，并将进度栏实例的 visible 属性设置为 false：

```
private function playNextVideo():void
{
  if (idx < (videosXML.length() - 1))
  {
    idx++;
    playVideo();
    positionBar.visible = true;
  }
  else
  {
    idx++;
    vid.clear();
    positionBar.visible = false;
  }
}
```

第 20 章

# 处理声音

20

ActionScript 是为开发引人入胜的交互式应用程序而设计的，这种极其引人入胜的应用程序中经常被忽略的一个元素是声音。您可以在视频游戏中添加声音效果，在应用程序用户界面中添加音频回馈，甚至创建一个分析通过 Internet 加载的 mp3 文件的程序（在这些情况下，声音是应用程序的核心）。

在本章中，您将了解如何加载外部音频文件以及处理 SWF 中嵌入的音频。您还会了解如何控制音频、创建声音信息的可视表示形式以及从用户麦克风捕获声音。

## 目录

# 声音处理基础知识

## 处理声音简介

就像计算机可以采用数字格式对图像进行编码、将它们存储在计算机上以及检索它们以便在屏幕上显示它们一样，计算机可以捕获并编码数字音频（声音信息的计算机表示形式）以及对其进行存储和检索，以通过连接到计算机上的扬声器进行回放。一种回放声音的方法是使用 Adobe Flash Player 和 ActionScript。

将声音数据转换为数字形式后，它具有各种不同的特性，如声音的音量以及它是立体声还是单声道声音。在 ActionScript 中回放声音时，您也可以调整这些特性；例如，使声音变得更大，或者使其像是来自某个方向。

在 ActionScript 中控制声音之前，您需要先将声音信息加载到 Flash Player 中。可以使用四种方法将音频数据加载到 Flash Player 中，以便通过 ActionScript 对其进行使用。您可以将外部声音文件（如 mp3 文件）加载到 SWF 中；在创建 SWF 文件时将声音信息直接嵌入到其中；使用连接到用户计算机上的麦克风来获取音频输入，以及访问从服务器流式传输的声音数据。

从外部声音文件加载声音数据时，您可以在仍加载其余声音数据的同时开始回放声音文件的开头部分。

虽然可以使用各种不同的声音文件格式对数字音频进行编码，但是 ActionScript 3.0 和 Flash Player 支持以 mp3 格式存储的声音文件。它们不能直接加载或播放其它格式的声音文件，如 WAV 或 AIFF。

在 ActionScript 中处理声音时，您可能会使用 flash.media 包中的某些类。通过使用 Sound 类，您可以加载声音文件并开始回放以获取对音频信息的访问。开始播放声音后， Flash Player 可为您提供对 SoundChannel 对象的访问。由于已加载的音频文件只能是您在用户计算机上播放的几种声音之一，因此，所播放的每种单独的声音使用其自己的 SoundChannel 对象；混合在一起的所有 SoundChannel 对象的组合输出是实际通过计算机扬声器播放的声音。可以使用此 SoundChannel 实例来控制声音的属性以及将其停止回放。最后，如果要控制组合音频，您可以通过 SoundMixer 类对混合输出进行控制。

也可以使用几个其它类，在 ActionScript 中处理声音时执行更具体的任务；要了解与声音有关的所有类的详细信息，请参阅第 516 页的 “了解声音体系结构”。

## 常见的声音处理任务

本章介绍了以下您将希望执行的与声音有关的任务：

- 加载外部 mp3 文件并跟踪其加载进度
- 播放、暂停、继续播放以及停止播放声音
- 在加载的同时播放声音流
- 控制音量和声相
- 从 mp3 文件中检索 ID3 元数据
- 使用原始声音波形数据
- 从用户麦克风捕获并重新播放声音输入

## 重要概念和术语

以下参考列表包含您将会在本章中遇到的重要术语：

- 波幅 (Amplitude)：声音波形上的点与零或平衡线之间的距离。
- 比特率 (Bit rate)：每秒为声音文件编码或流式传输的数据量。对于 mp3 文件，比特率通常是以每秒千位数 (kbps) 来表述的。较高的比特率通常意味着较高品质的声音波形。
- 缓冲 (Buffering)：在回放之前接收和存储声音数据。
- mp3：MPEG-1 Audio Layer 3 (mp3) 是一种常用的声音压缩格式。
- 声相 (Panning)：将音频信号放在立体声声场中左声道和右声道之间。
- 峰值 (Peak)：波形中的最高点。
- 采样率 (Sampling rate)：定义在生成数字信号时每秒从模拟音频信号采集的样本数。标准光盘音频的采样率为 44.1 kHz 或每秒 44,100 个样本。
- 流式传输 (Streaming)：此过程是指，在仍从服务器加载声音文件或视频文件的后面部分的同时播放该文件的前面部分。
- 音量 (Volume)：声音的响度。
- 波形 (Waveform)：声音信号波幅随时间变化的图形形状。

## 完成本章中的示例

学习本章的过程中，您可能希望测试某些示例代码清单。由于本章介绍了在 ActionScript 中处理声音，因此许多示例都涉及对声音文件的处理，例如，播放声音文件、停止回放或以某种方式调整声音。测试本章中的示例：

1. 创建一个新的 Flash 文档并将它保存到您的计算机上。
2. 在时间轴上，选择第一个关键帧，然后打开 “动作” 面板。
3. 将示例代码清单复制到 “脚本” 窗格中。

4. 如果代码涉及加载外部声音文件，则其中会有一行与下面的代码类似的代码：

   ```
   var req:URLRequest = new URLRequest("click.mp3");
   var s:Sound = new Sound(req);
   ```

   其中 “click.mp3” 是要加载的声音文件的名称。为了测试这些示例，需要有一个可使用的 mp3 文件。应将此 mp3 文件放入 Flash 文档所在的文件夹中。然后应更改代码以使用 mp3 文件的名称，而不是代码清单中的名称（例如，在上面的代码中，将 “click.mp3” 改为 mp3 文件的名称）。

5. 从主菜单中，选择 “控制” > “测试影片” 以创建 SWF 文件并预览 （并且听一下）示例的输出。

除了播放音频外，某些示例将使用 `trace()` 函数来显示值；当测试这些示例时，将会在 “输出” 面板中看到这些值的结果。一些示例还将内容绘制到屏幕上，因此对于这些示例，您还将在 Flash Player 窗口中看到内容。

有关测试本手册中的示例代码清单的详细信息，请参阅第 53 页的 “测试本章内的示例代码清单”。

# 了解声音体系结构

应用程序可以从以下四种主要来源加载声音数据：

- 在运行时加载的外部声音文件
- 在应用程序的 SWF 文件中嵌入的声音资源
- 来自连接到用户系统上的麦克风的声音数据
- 从远程媒体服务器流式传输的声音数据，如 Flash Media Server

可以在回放之前完全加载声音数据，也可以进行流式传输，即在仍进行加载的同时回放这些数据。

ActionScript 3.0 和 Flash Player 支持以 mp3 格式存储的声音文件。它们不能直接加载或播放其它格式的声音文件，如 WAV 或 AIFF。

使用 Adobe Flash CS3 Professional，可以导入 WAV 或 AIFF 声音文件，然后将其以 mp3 格式嵌入应用程序的 SWF 文件中。 Flash 创作工具还可压缩嵌入的声音文件以减小文件大小，但会降低声音的品质。有关详细信息，请参阅 《使用 Flash》中的 “导入声音”。

ActionScript 3.0 声音体系结构使用 flash.media 包中的以下类。

| 类 | 描述 |
|---|---|
| flash.media.Sound | Sound 类处理声音加载、管理基本声音属性以及启动声音播放。 |
| flash.media.SoundChannel | 当应用程序播放 Sound 对象时，将创建一个新的 SoundChannel 对象来控制回放。 SoundChannel 对象控制声音的左和右回放声道的音量。播放的每种声音具有其自己的 SoundChannel 对象。 |
| flash.media.SoundLoaderContext | SoundLoaderContext 类指定在加载声音时使用的缓冲秒数，以及 Flash Player 在加载文件时是否从服务器中查找跨域策略文件。 SoundLoaderContext 对象用作 `Sound.load()` 方法的参数。 |
| flash.media.SoundMixer | SoundMixer 类控制与应用程序中的所有声音有关的回放和安全属性。实际上，可通过一个通用 SoundMixer 对象将多个声道混合在一起，因此，该 SoundMixer 对象中的属性值将影响当前播放的所有 SoundChannel 对象。 |
| flash.media.SoundTransform | SoundTransform 类包含控制音量和声相的值。可以将 SoundTransform 对象应用于单个 SoundChannel 对象、全局 SoundMixer 对象或 Microphone 对象等。 |
| flash.media.ID3Info | ID3Info 对象包含一些属性，它们表示通常存储在 mp3 声音文件中的 ID3 元数据信息。 |
| flash.media.Microphone | Microphone 类表示连接到用户计算机上的麦克风或其它声音输入设备。可以将来自麦克风的音频输入传送到本地扬声器或发送到远程服务器。 Microphone 对象控制其自己的声音流的增益、采样率以及其它特性。 |

加载和播放的每种声音需要其自己的 Sound 类和 SoundChannel 类的实例。然后，全局 SoundMixer 类在回放期间将来自多个 SoundChannel 实例的输出混合在一起。

Sound、 SoundChannel 和 SoundMixer 类不能用于从麦克风或流媒体服务器 （如 Flash Media Server）中获取的声音数据。

# 加载外部声音文件

Sound 类的每个实例可加载并触发特定声音资源的回放。应用程序无法重复使用 Sound 对象来加载多种声音。如果它要加载新的声音资源，则应创建一个新的 Sound 对象。

如果要加载较小的声音文件 （如要附加到按钮上的单击声音），应用程序可以创建一个新的 Sound，并让其自动加载该声音文件，如下所示：

```
var req:URLRequest = new URLRequest("click.mp3");
var s:Sound = new Sound(req);
```

Sound() 构造函数接受一个 URLRequest 对象作为其第一个参数。当提供 URLRequest 参数的值后，新的 Sound 对象将自动开始加载指定的声音资源。

除了最简单的情况下，应用程序都应关注声音的加载进度，并监视在加载期间出现的错误。例如，如果单击声音非常大，在用户单击触发该声音的按钮时，该声音可能没有完全加载。尝试播放未加载的声音可能会导致运行时错误。较为稳妥的作法是等待声音完全加载后，再让用户执行可能启动声音播放的动作。

Sound 对象将在声音加载过程中调度多种不同的事件。应用程序可以侦听这些事件以跟踪加载进度，并确保在播放之前完全加载声音。下表列出了可以由 Sound 对象调度的事件。

| 事件 | 描述 |
| --- | --- |
| open (Event.OPEN) | 就在声音加载操作开始之前进行调度。 |
| progress (ProgressEvent.PROGRESS) | 从文件或流接收数据时，在声音加载过程中定期进行调度。 |
| id3 (Event.ID3) | 当存在可用于 mp3 声音的 ID3 数据时进行调度。 |
| complete (Event.COMPLETE) | 在加载了所有声音资源后进行调度。 |
| ioError (IOErrorEvent.IO_ERROR) | 在以下情况下进行调度：找不到声音文件，或者在收到所有声音数据之前加载过程中断。 |

以下代码说明了如何在完成加载后播放声音：

```
import flash.events.Event;
import flash.media.Sound;
import flash.net.URLRequest;

var s:Sound = new Sound();
s.addEventListener(Event.COMPLETE, onSoundLoaded);
var req:URLRequest = new URLRequest("bigSound.mp3");
s.load(req);

function onSoundLoaded(event:Event):void
{
    var localSound:Sound = event.target as Sound;
    localSound.play();
}
```

首先，该代码范例创建一个新的 Sound 对象，但没有为其指定 URLRequest 参数的初始值。然后，它通过 Sound 对象侦听 Event.COMPLETE 事件，该对象导致在加载完所有声音数据后执行 onSoundLoaded() 方法。接下来，它使用新的 URLRequest 值为声音文件调用 Sound.load() 方法。

在加载完声音后，将执行 onSoundLoaded() 方法。Event 对象的目标属性是对 Sound 对象的引用。如果调用 Sound 对象的 play() 方法，则会启动声音回放。

## 监视声音加载过程

声音文件可能很大，而需要花很长时间进行加载。尽管 Flash Player 允许应用程序甚至在完全加载声音之前播放声音，但您可能需要向用户指示已加载了多少声音数据以及已播放了多少声音。

Sound 类调度以下两个事件，它们可使声音加载进度显示变得相对比较简单：ProgressEvent.PROGRESS 和 Event.COMPLETE。以下示例说明了如何使用这些事件来显示有关所加载的声音的进度信息：

```
import flash.events.Event;
import flash.events.ProgressEvent;
import flash.media.Sound;
import flash.net.URLRequest;

var s:Sound = new Sound();
s.addEventListener(ProgressEvent.PROGRESS, onLoadProgress);
s.addEventListener(Event.COMPLETE, onLoadComplete);
s.addEventListener(IOErrorEvent.IO_ERROR, onIOError);

var req:URLRequest = new URLRequest("bigSound.mp3");
s.load(req);

function onLoadProgress(event:ProgressEvent):void
{
  var loadedPct:uint =
    Math.round(100 * (event.bytesLoaded / event.bytesTotal));
  trace("The sound is " + loadedPct + "% loaded.");
}

function onLoadComplete(event:Event):void
{
  var localSound:Sound = event.target as Sound;
  localSound.play();
}
function onIOError(event:IOErrorEvent)
{
  trace("The sound could not be loaded: " + event.text);
}
```

此代码先创建一个 Sound 对象，然后在该对象中添加侦听器以侦听 ProgressEvent.PROGRESS 和 Event.COMPLETE 事件。在调用 Sound.load() 方法并从声音文件接收第一批数据后，将会发生 ProgressEvent.PROGRESS 事件并触发 onSoundLoadProgress() 方法。

已加载的声音数据百分比等于 ProgressEvent 对象的 bytesLoaded 属性值除以 bytesTotal 属性值。 Sound 对象上也提供了相同的 bytesLoaded 和 bytesTotal 属性。以上示例只显示了有关声音加载进度的消息，但您可以方便地使用 bytesLoaded 和 bytesTotal 值来更新进度栏组件，例如随 Adobe Flex 2 框架或 Flash 创作工具提供的组件。

此示例还说明了应用程序在加载声音文件时如何识别并响应出现的错误。例如，如果找不到具有给定文件名的声音文件，Sound 对象将调度一个 `Event.IO_ERROR` 事件。在上面的代码中，当发生错误时，将执行 `onIOError()` 方法并显示一条简短的错误消息。

# 处理嵌入的声音

对于用作应用程序用户界面中的指示器的较小声音 （如在单击按钮时播放的声音），使用嵌入的声音非常有用 （而不是从外部文件加载声音）。

在应用程序中嵌入声音文件时，生成的 SWF 文件大小比原来增加了声音文件的大小。也就是说，如果在应用程序中嵌入较大的声音文件，可能会使 SWF 文件增大到难以接受的大小。

将声音文件嵌入到应用程序的 SWF 文件中的具体方法因开发环境而异。

## 在 Flash 中使用嵌入的声音文件

Flash 创作工具可导入多种声音格式的声音并将其作为元件存储在库中。然后，您可以将其分配给时间轴上的帧或按钮状态的帧，通过行为来使用声音，或直接在 ActionScript 代码中使用它们。本节说明如何在 ActionScript 代码中通过 Flash 创作工具来使用嵌入的声音。有关在 Flash 中使用嵌入的声音的其它方法，请参阅 《使用 Flash》中的 “导入声音”。

**要在 Flash 影片中嵌入声音文件，请执行以下操作：**

1. 选择 “文件” > “导入” > “导入到库”，然后选择一个声音文件并导入它。
2. 在“库”面板中，右键单击导入的文件的名称，然后选择“属性”。单击“为 ActionScript 导出”复选框。
3. 在 “类”字段中，输入一个名称，以便在 ActionScript 中引用此嵌入的声音时使用。默认情况下，它将使用此字段中声音文件的名称。如果文件名包含句点 （如名称 “DrumSound.mp3”），则必须将其更改为类似于 “DrumSound”这样的名称；ActionScript 不允许在类名称中出现句点字符。“基类”字段应仍显示 flash.media.Sound。
4. 单击 “确定”。可能出现一个对话框，指出无法在类路径中找到该类的定义。单击 “确定”以继续。如果输入的类名称与应用程序的类路径中任何类的名称都不匹配，则会自动生成从 flash.media.Sound 类继承的新类。
5. 要使用嵌入的声音，请在 ActionScript 中引用该声音的类名称。例如，通过创建自动生成的 DrumSound 类的一个新实例来启动以下代码：

   ```
   var drum:DrumSound = new DrumSound();
   var channel:SoundChannel = drum.play();
   ```

   DrumSound 是 flash.media.Sound 类的子类，所以它继承了 Sound 类的方法和属性，包括上面显示的 `play()` 方法。

# 处理声音流文件

如果在仍加载声音文件或视频文件数据的同时回放该文件，则认为是*流式传输*。通常，将对从远程服务器加载的外部声音文件进行流式传输，以使用户不必等待加载完所有声音数据再收听声音。

`SoundMixer.bufferTime` 属性表示 Flash Player 在允许播放声音之前应收集多长时间的声音数据（以毫秒为单位）。也就是说，如果将 `bufferTime` 属性设置为 5000，在开始播放声音之前，Flash Player 将从声音文件中加载至少相当于 5000 毫秒的数据。`SoundMixer.bufferTime` 默认值为 1000。

通过在加载声音时显式地指定新的 `bufferTime` 值，应用程序可以覆盖单个声音的全局 `SoundMixer.bufferTime` 值。要覆盖默认缓冲时间，请先创建一个新的 SoundLoaderContext 类实例，设置其 `bufferTime` 属性，然后将其作为参数传递给 `Sound.load()` 方法，如下所示：

```
import flash.media.Sound;
import flash.media.SoundLoaderContext;
import flash.net.URLRequest;

var s:Sound = new Sound();
var req:URLRequest = new URLRequest("bigSound.mp3");
var context:SoundLoaderContext = new SoundLoaderContext(8000, true);
s.load(req, context);
s.play();
```

当回放继续进行时，Flash Player 尝试将声音缓冲保持在相同大小或更大。如果声音数据的加载速度比回放快，回放将继续进行而不会中断。但是，如果数据加载速率由于网络限制而减慢，播放头可能会到达声音缓冲区的结尾。如果发生这种情况，将暂停回放，但会在加载更多声音数据后自动恢复回放。

要查明暂停回放是否是由于 Flash Player 正在等待加载数据，请使用 `Sound.isBuffering` 属性。

# 播放声音

播放加载的声音非常简便，您只需为 Sound 对象调用 `Sound.play()` 方法，如下所示：

```
var snd:Sound = new Sound(new URLRequest("smallSound.mp3"));
snd.play();
```

使用 ActionScript 3.0 回放声音时，您可以执行以下操作：

- 从特定起始位置播放声音
- 暂停声音并稍后从相同位置恢复回放
- 准确了解何时播放完声音

■ 跟踪声音的回放进度
■ 在播放声音的同时更改音量或声相

要在回放期间执行这些操作，请使用 SoundChannel、SoundMixer 和 SoundTransform 类。

SoundChannel 类控制一种声音的回放。可以将 SoundChannel.position 属性视为播放头，以指示所播放的声音数据中的当前位置。

当应用程序调用 Sound.play() 方法时，将创建一个新的 SoundChannel 类实例来控制回放。

通过将特定起始位置（以毫秒为单位）作为 Sound.play() 方法的 startTime 参数进行传递，应用程序可以从该位置播放声音。它也可以通过在 Sound.play() 方法的 loops 参数中传递一个数值，指定快速且连续地将声音重复播放固定的次数。

使用 startTime 参数和 loops 参数调用 Sound.play() 方法时，每次将从相同的起始点重复回放声音，如以下代码中所示：

```
var snd:Sound = new Sound(new URLRequest("repeatingSound.mp3"));
snd.play(1000, 3);
```

在此示例中，从声音开始后的 1 秒起连续播放声音三次。

## 暂停和恢复播放声音

如果应用程序播放很长的声音（如歌曲或播客），您可能需要让用户暂停和恢复回放这些声音。实际上，无法在 ActionScript 中的回放期间暂停声音；而只能将其停止。但是，可以从任何位置开始播放声音。您可以记录声音停止时的位置，并随后从该位置开始重放声音。

例如，假定代码加载并播放一个声音文件，如下所示：

```
var snd:Sound = new Sound(new URLRequest("bigSound.mp3"));
var channel:SoundChannel = snd.play();
```

在播放声音的同时，SoundChannel.position 属性指示当前播放到的声音文件位置。应用程序可以在停止播放声音之前存储位置值，如下所示：

```
var pausePosition:int = channel.position;
channel.stop();
```

要恢复播放声音，请传递以前存储的位置值，以便从声音以前停止的相同位置重新启动声音。

```
channel = snd.play(pausePosition);
```

## 监视回放

应用程序可能需要了解何时停止播放某种声音，以便开始播放另一种声音，或者清除在以前回放期间使用的某些资源。SoundChannel 类在其声音完成播放时将调度 Event.SOUND_COMPLETE 事件。应用程序可以侦听此事件并执行相应的动作，如下所示：

```
import flash.events.Event;
import flash.media.Sound;
import flash.net.URLRequest;

var snd:Sound = new Sound("smallSound.mp3");
var channel:SoundChannel = snd.play();
s.addEventListener(Event.SOUND_COMPLETE, onPlaybackComplete);

public function onPlaybackComplete(event:Event)
{
  trace("The sound has finished playing.");
}
```

SoundChannel 类在回放期间不调度进度事件。要报告回放进度，应用程序可以设置其自己的计时机制并跟踪声音播放头的位置。

要计算已播放的声音百分比，您可以将 SoundChannel.position 属性值除以所播放的声音数据长度：

```
var playbackPercent:uint = 100 * (channel.position / snd.length);
```

但是，仅当在开始回放之前完全加载了声音数据，此代码才会报告精确的回放百分比。Sound.length 属性显示当前加载的声音数据的大小，而不是整个声音文件的最终大小。要跟踪仍在加载的声音流的回放进度，应用程序应估计完整声音文件的最终大小，并在其计算中使用该值。您可以使用 Sound 对象的 bytesLoaded 和 bytesTotal 属性来估计声音数据的最终长度，如下所示：

```
var estimatedLength:int =
  Math.ceil(snd.length / (snd.bytesLoaded / snd.bytesTotal));
var playbackPercent:uint = 100 * (channel.position / estimatedLength);
```

以下代码加载一个较大的声音文件，并使用 Event.ENTER_FRAME 事件作为其计时机制来显示回放进度。它定期报告回放百分比，这是作为当前位置值除以声音数据的总长度来计算的：

```
import flash.events.Event;
import flash.media.Sound;
import flash.net.URLRequest;

var snd:Sound = new Sound();
var req:URLRequest = new
  URLRequest("http://av.adobe.com/podcast/csbu_dev_podcast_epi_2.mp3");
snd.load(req);

var channel:SoundChannel;
channel = snd.play();
```

```
addEventListener(Event.ENTER_FRAME, onEnterFrame);
snd.addEventListener(Event.SOUND_COMPLETE, onPlaybackComplete);

function onEnterFrame(event:Event):void
{
  var estimatedLength:int =
    Math.ceil(snd.length / (snd.bytesLoaded / snd.bytesTotal));
  var playbackPercent:uint =
    Math.round(100 * (channel.position / estimatedLength));
  trace("Sound playback is " + playbackPercent + "% complete.");
}

function onPlaybackComplete(event:Event)
{
  trace("The sound has finished playing.");
  removeEventListener(Event.ENTER_FRAME, onEnterFrame);
}
```

在开始加载声音数据后，此代码调用 snd.play() 方法，并将生成的 SoundChannel 对象存储在 channel 变量中。随后，此代码在主应用程序中添加 Event.ENTER_FRAME 事件的事件侦听器，并在 SoundChannel 对象中添加另一个事件侦听器，用于侦听在回放完成时发生的 Event.SOUND_COMPLETE 事件。

每次应用程序到达其动画中的新帧时，将调用 onEnterFrame() 方法。onEnterFrame() 方法基于已加载的数据量来估计声音文件的总长度，然后计算并显示当前回放百分比。

当播放整个声音后，将执行 onPlaybackComplete() 方法来删除 Event.ENTER_FRAME 事件的事件侦听器，以使其在完成回放后不会尝试显示进度更新。

可以每秒多次调度 Event.ENTER_FRAME 事件。在某些情况下，您不需要频繁显示回放进度。在这些情况下，应用程序可以使用 flash.util.Timer 类来设置其自己的计时机制；请参阅 第 159 页的 “处理日期和时间”。

## 停止声音流

在进行流式传输的声音（即，在播放的同时仍在加载声音）的回放过程中，有一个奇怪的现象。当应用程序对回放声音流的 SoundChannel 实例调用 SoundChannel.stop() 方法时，声音回放在一个帧处停止，随后在下一帧处从声音开头重新回放。发生这种情况是因为，声音加载过程仍在进行当中。要停止声音流加载和回放，请调用 Sound.close() 方法。

# 加载和播放声音时的安全注意事项

可以根据 Flash Player 安全模型来限制应用程序访问声音数据的功能。每种声音受两种不同的安全沙箱的限制：内容本身的沙箱 （“内容沙箱”）以及加载和播放声音的应用程序或对象的沙箱 （“所有者沙箱”）。要了解有关常用 Flash Player 安全模型的详细信息以及沙箱定义，请参阅第 649 页的 “Flash Player 安全性”。

内容沙箱控制使用 `id3` 属性还是 `SoundMixer.computeSpectrum()` 方法从声音提取详细声音数据。它不会限制声音文件本身的加载或播放。

声音文件的原始域定义了内容沙箱的安全限制。一般来说，如果某个声音文件与加载该文件的应用程序或对象的 SWF 文件位于相同的域或文件夹中，则应用程序或对象具有该声音文件的完全访问权限。如果声音来自与应用程序不同的域，仍可以使用跨域策略文件将其加载到内容沙箱中。

应用程序可以将带有 `checkPolicyFile` 属性的 SoundLoaderContext 对象作为参数传递给 `Sound.load()` 方法。如果将 `checkPolicyFile` 属性设置为 `true`，则会通知 Flash Player 在从中加载声音的服务器上查找跨域策略文件。如果存在跨域策略文件，并且它为执行加载的 SWF 文件所在的域授予了访问权限，则该 SWF 文件可以加载声音文件、访问 Sound 对象的 `id3` 属性以及为加载的声音调用 `SoundMixer.computeSpectrum()` 方法。

所有者沙箱控制声音的本地回放。所有者沙箱是由开始播放声音的应用程序或对象定义的。

只要当前播放的所有 SoundChannel 对象中的声音符合以下条件，`SoundMixer.stopAll()` 方法就会将它们静音：

- 声音是由相同所有者沙箱中的对象启动的。
- 声音来自具有跨域策略文件 （为调用 `SoundMixer.stopAll()` 方法的应用程序或对象所在的域授予访问权限）的源。

要查明 `SoundMixer.stopAll()` 方法是否确实停止了所有播放的声音，应用程序可以调用 `SoundMixer.areSoundsInaccessible()` 方法。如果该方法返回值 `true`，则当前所有者沙箱无法控制播放的某些声音， `SoundMixer.stopAll()` 方法不会将其停止。

`SoundMixer.stopAll()` 方法还会阻止播放头继续播放从外部文件加载的所有声音。但是，如果动画移动到一个新帧， FLA 文件中嵌入的声音以及使用 Flash 创作工具附加到时间轴中的帧上的声音可能会重新开始播放。

# 控制音量和声相

单个 SoundChannel 对象控制声音的左和右立体声声道。如果 mp3 声音是单声道声音，SoundChannel 对象的左和右立体声声道将包含完全相同的波形。

可通过使用 SoundChannel 对象的 `leftPeak` 和 `rightPeak` 属性来查明所播放的声音的每个立体声声道的波幅。这些属性显示声音波形本身的峰值波幅。它们并不表示实际回放音量。实际回放音量是声音波形的波幅以及 SoundChannel 对象和 SoundMixer 类中设置的音量值的函数。

在回放期间，可以使用 SoundChannel 对象的 pan 属性为左和右声道分别指定不同的音量级别。 pan 属性可以具有范围从 -1 到 1 的值，其中， -1 表示左声道以最大音量播放，而右声道处于静音状态； 1 表示右声道以最大音量播放，而左声道处于静音状态。介于 -1 和 1 之间的数值为左和右声道值设置一定比例的值，值 0 表示两个声道以均衡的中音量级别播放。

以下代码示例使用 volume 值 0.6 和 pan 值 -1 创建一个 SoundTransform 对象 （左声道为最高音量，右声道没有音量） 。此代码将 SoundTransform 对象作为参数传递给 `play()` 方法，此方法将该 SoundTransform 对象应用于为控制回放而创建的新 SoundChannel 对象。

```
var snd:Sound = new Sound(new URLRequest("bigSound.mp3"));
var trans:SoundTransform = new SoundTransform(0.6, -1);
var channel:SoundChannel = snd.play(0, 1, trans);
```

可以在播放声音的同时更改音量和声相，方法是：设置 SoundTransform 对象的 `pan` 或 `volume` 属性，然后将该对象作为 SoundChannel 对象的 `soundTransform` 属性进行应用。

也可以通过使用 SoundMixer 类的 `soundTransform` 属性，同时为所有声音设置全局音量和声相值，如以下示例所示：

```
SoundMixer.soundTransform = new SoundTransform(1, -1);
```

也可以使用 SoundTransform 对象为 Microphone 对象 （请参阅第 532 页的 “捕获声音输入”）、 Sprite 对象和 SimpleButton 对象设置音量和声相值。

以下示例在播放声音的同时将声音从左声道移到右声道，然后再移回来，并交替进行这一过程。

```
import flash.events.Event;
import flash.media.Sound;
import flash.media.SoundChannel;
import flash.media.SoundMixer;
import flash.net.URLRequest;

var snd:Sound = new Sound();
var req:URLRequest = new URLRequest("bigSound.mp3");
snd.load(req);

var panCounter:Number = 0;

var trans:SoundTransform;
```

```
trans = new SoundTransform(1, 0);
var channel:SoundChannel = snd.play(0, 1, trans);
channel.addEventListener(Event.SOUND_COMPLETE, onPlaybackComplete);

addEventListener(Event.ENTER_FRAME, onEnterFrame);

function onEnterFrame(event:Event):void
{
  trans.pan = Math.sin(panCounter);
  channel.soundTransform = trans; // or SoundMixer.soundTransform = trans;
  panCounter += 0.05;
}

function onPlaybackComplete(event:Event):void
{
  removeEventListener(Event.ENTER_FRAME, onEnterFrame);
}
```

此代码先加载一个声音文件，然后将 volume 设置为 1（最大音量）并将 pan 设置为 0（声音在左和右声道之间均衡地平均分布）以创建一个新的 SoundTransform 对象。接下来，此代码调用 `snd.play()` 方法，以将 SoundTransform 对象作为参数进行传递。

在播放声音时，将反复执行 `onEnterFrame()` 方法。`onEnterFrame()` 方法使用 `Math.sin()` 函数来生成介于 -1 和 1 之间的值，此范围对应于可接受的 `SoundTransform.pan` 属性值。此代码将 SoundTransform 对象的 `pan` 属性设置为新值，然后设置声道的 `soundTransform` 属性以使用更改后的 SoundTransform 对象。

要运行此示例，请用本地 mp3 文件的名称替换文件名 bigSound.mp3。然后，运行该示例。当右声道音量变小时，您应会听到左声道音量变大，反之亦然。

在此示例中，可通过设置 SoundMixer 类的 `soundTransform` 属性来获得同样的效果。但是，这会影响当前播放的所有声音的声相，而不是只影响此 SoundChannel 对象播放的一种声音。

# 处理声音元数据

使用 mp3 格式的声音文件可以采用 ID3 标签格式来包含有关声音的其它数据。

并非每个 mp3 文件都包含 ID3 元数据。当 Sound 对象加载 mp3 声音文件时，如果该声音文件包含 ID3 元数据，它将调度 `Event.ID3` 事件。要防止出现运行时错误，应用程序应等待接收 `Event.ID3` 事件后，再访问加载的声音的 `Sound.id3` 属性。

以下代码说明了如何识别何时加载了声音文件的 ID3 元数据：

```
import flash.events.Event;
import flash.media.ID3Info;
import flash.media.Sound;
```

```
var s:Sound = new Sound();
s.addEventListener(Event.ID3, onID3InfoReceived);
s.load("mySound.mp3");

function onID3InfoReceived(event:Event)
{
  var id3:ID3Info = event.target.id3;

  trace("Received ID3 Info:");
  for (var propName:String in id3)
  {
    trace(propName + " = " + id3[propName]);
  }
}
```

此代码先创建一个 Sound 对象并通知它侦听 `Event.ID3` 事件。加载了声音文件的 ID3 元数据后，将调用 `onID3InfoReceived()` 方法。传递给 `onID3InfoReceived()` 方法的 Event 对象的目标是原始 Sound 对象，因此，该方法随后获取 Sound 对象的 `id3` 属性，然后循环访问所有命名属性以跟踪它们的值。

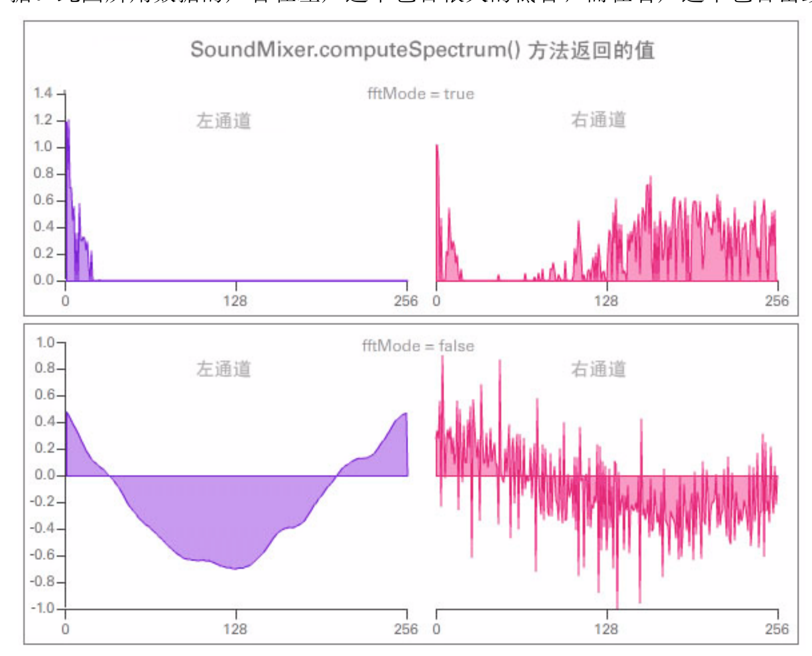
# 访问原始声音数据

通过使用 `SoundMixer.computeSpectrum()` 方法，应用程序可以读取当前所播放的波形的原始声音数据。如果当前播放多个 SoundChannel 对象，`SoundMixer.computeSpectrum()` 方法将显示混合在一起的每个 SoundChannel 对象的组合声音数据。

声音数据是作为 ByteArray 对象 （包含 512 个字节的数据）返回的，其中的每个字节包含一个介于 -1 和 1 之间的浮点值。这些值表示所播放的声音波形中的点的波幅。这些值是分为两个组 （每组包含 256 个值）提供的，第一个组用于左立体声声道，第二个组用于右立体声声道。

如果将 `FFTMode` 参数设置为 `true`， `SoundMixer.computeSpectrum()` 方法将返回频谱数据，而非波形数据。频谱显示按声音频率 （从最低频率到最高频率）排列的波幅。可以使用快速傅立叶变换 (FFT) 将波形数据转换为频谱数据。生成的频谱值范围介于 0 和约 1.414 （2 的平方根）之间。

下图比较了将 FFTMode 参数设置为 true 和 false 时从 computeSpectrum() 方法返回的数据。此图所用数据的声音在左声道中包含很大的低音；而在右声道中包含击鼓声。




computeSpectrum() 方法也可以返回已在较低比特率重新采样的数据。通常，这会产生更平滑的波形数据或频率数据，但会以牺牲细节为代价。 stretchFactor 参数控制 computeSpectrum() 方法数据的采样率。如果将 stretchFactor 参数设置为 0 （默认值），则以采样率 44.1 kHz 采集声音数据样本。 stretchFactor 参数值每连续增加 1，采样率就减小一半，因此，值 1 指定采样率 22.05 kHz，值 2 指定采样率 11.025 kHz，依此类推。当使用较高的 stretchFactor 值时， computeSpectrum() 方法仍会为每个立体声声道返回 256 个字节。

SoundMixer.computeSpectrum() 方法具有一些限制：

- 由于来自麦克风或 RTMP 流的声音数据不是通过全局 SoundMixer 对象传递的，因此， SoundMixer.computeSpectrum() 方法不会从这些源返回数据。
- 如果播放的一种或多种声音来自当前内容沙箱以外的源，安全限制将导致 SoundMixer.computeSpectrum() 方法引发错误。有关 SoundMixer.computeSpectrum() 方法的安全限制的更多详细信息，请参阅第 525 页的 “加载和播放声音时的安全注意事项”和第 671 页的 “作为数据访问加载的媒体”。

## 构建简单的声音可视化程序

以下示例使用 SoundMixer.computeSpectrum() 方法来显示声音波形图（它对每个帧进行动画处理）：

```
import flash.display.Graphics;
import flash.events.Event;
import flash.media.Sound;
import flash.media.SoundChannel;
import flash.media.SoundMixer;
import flash.net.URLRequest;

const PLOT_HEIGHT:int = 200;
const CHANNEL_LENGTH:int = 256;

var snd:Sound = new Sound();
var req:URLRequest = new URLRequest("bigSound.mp3");
snd.load(req);

var channel:SoundChannel;
channel = snd.play();
addEventListener(Event.ENTER_FRAME, onEnterFrame);
snd.addEventListener(Event.SOUND_COMPLETE, onPlaybackComplete);

var bytes:ByteArray = new ByteArray();

function onEnterFrame(event:Event):void
{
  SoundMixer.computeSpectrum(bytes, false, 0);

  var g:Graphics = this.graphics;

  g.clear();
  g.lineStyle(0, 0x6600CC);
  g.beginFill(0x6600CC);
  g.moveTo(0, PLOT_HEIGHT);

  var n:Number = 0;

  // left channel
  for (var i:int = 0; i < CHANNEL_LENGTH; i++)
  {
    n = (bytes.readFloat() * PLOT_HEIGHT);
    g.lineTo(i * 2, PLOT_HEIGHT - n);
  }
  g.lineTo(CHANNEL_LENGTH * 2, PLOT_HEIGHT);
  g.endFill();

  // right channel
  g.lineStyle(0, 0xCC0066);
```

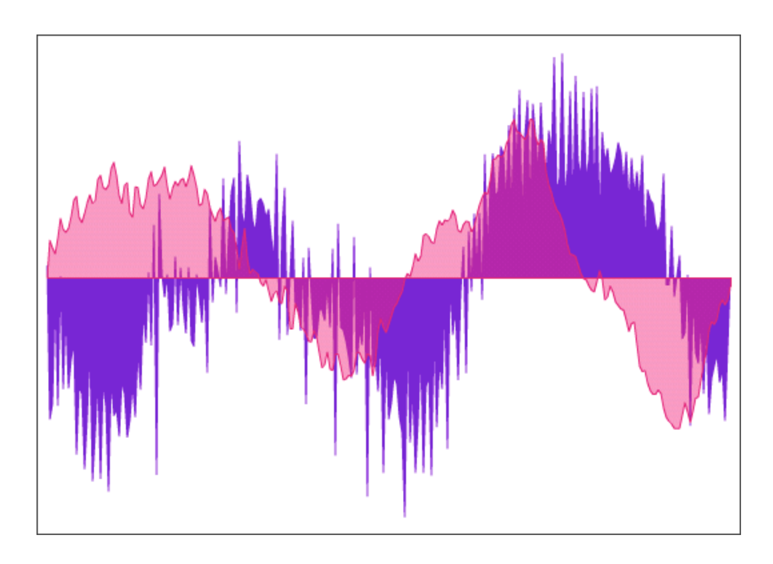
```
    g.beginFill(0xCC0066, 0.5);
    g.moveTo(CHANNEL_LENGTH * 2, PLOT_HEIGHT);

    for (i = CHANNEL_LENGTH; i > 0; i--)
    {
        n = (bytes.readFloat() * PLOT_HEIGHT);
        g.lineTo(i * 2, PLOT_HEIGHT - n);
    }
    g.lineTo(0, PLOT_HEIGHT);
    g.endFill();
}

function onPlaybackComplete(event:Event)
{
    removeEventListener(Event.ENTER_FRAME, onEnterFrame);
}
```

此示例先加载并播放一个声音文件，然后在播放声音的同时侦听将触发 `onEnterFrame()` 方法的 `Event.ENTER_FRAME` 事件。`onEnterFrame()` 方法先调用 `SoundMixer.computeSpectrum()` 方法，后者将声音波形数据存储在 `bytes` ByteArray 对象中。

声音波形是使用矢量绘图 API 绘制的。`for` 循环将循环访问第一批 256 个数据值（表示左立体声声道），然后使用 `Graphics.lineTo()` 方法绘制一条从每个点到下一个点的直线。第二个 `for` 循环将循环访问下一批 256 个值，此时按相反的顺序（从右到左）对它们进行绘制。生成的波形图可能会产生非常有趣的镜像图像效果，如以下图像中所示。



# 捕获声音输入

应用程序可通过 Microphone 类连接到用户系统上的麦克风或其它声音输入设备，并将输入音频广播到该系统的扬声器，或者将音频数据发送到远程服务器，如 Flash Media Server。

## 访问麦克风

Microphone 类没有构造函数方法。相反，应使用静态 `Microphone.getMicrophone()` 方法来获取新的 Microphone 实例，如下所示：

```
var mic:Microphone = Microphone.getMicrophone();
```

不使用参数调用 `Microphone.getMicrophone()` 方法时，将返回在用户系统上发现的第一个声音输入设备。

系统可能连接了多个声音输入设备。应用程序可以使用 `Microphone.names` 属性来获取所有可用声音输入设备名称的数组。然后，它可以使用 `index` 参数（与数组中的设备名称的索引值相匹配）来调用 `Microphone.getMicrophone()` 方法。

系统可能没有连接麦克风或其它声音输入设备。可以使用 `Microphone.names` 属性或 `Microphone.getMicrophone()` 方法来检查用户是否安装了声音输入设备。如果用户未安装声音输入设备，则 `names` 数组的长度为零，并且 `getMicrophone()` 方法返回值 `null`。

当应用程序调用 `Microphone.getMicrophone()` 方法时， Flash Player 将显示 "Flash Player 设置" 对话框，它提示用户允许或拒绝 Flash Player 对系统上的摄像头和麦克风的访问。在用户单击此对话框中的 "允许" 或 "拒绝" 按钮后，将调度 StatusEvent。该 StatusEvent 实例的 `code` 属性指示是允许还是拒绝对麦克风的访问，如下例所示：

```
import flash.media.Microphone;

var mic:Microphone = Microphone.getMicrophone();
mic.addEventListener(StatusEvent.STATUS, this.onMicStatus);

function onMicStatus(event:StatusEvent):void
{
  if (event.code == "Microphone.Unmuted")
  {
    trace("Microphone access was allowed.");
  }
  else if (event.code == "Microphone.Muted")
  (
    trace("Microphone access was denied.");
  }
}
```

如果允许访问，`StatusEvent.code` 属性将包含 “Microphone.Unmuted”；如果拒绝访问，则包含 “Microphone.Muted”。

> **注意** 当用户允许或拒绝对麦克风的访问时，`Microphone.muted` 属性将被分别设置为 `true` 或 `false`。但是，在调度 StatusEvent 之前，不会在 Microphone 实例上设置 `muted` 属性，因此，应用程序还应等待调度 `StatusEvent.STATUS` 事件后再检查 `Microphone.muted` 属性。

## 将麦克风音频传送到本地扬声器

可以使用参数值 `true` 调用 `Microphone.setLoopback()` 方法，以将来自麦克风的音频输入传送到本地系统扬声器。

如果将来自本地麦克风的声音传送到本地扬声器，则会存在创建音频回馈循环的风险，这可能会导致非常大的振鸣声，并且可能会损坏声音硬件。使用参数值 `true` 调用 `Microphone.setUseEchoSuppression()` 方法可降低发生音频回馈的风险，但不会完全消除该风险。 Adobe 建议您始终在调用 `Microphone.setLoopback(true)` 之前调用 `Microphone.setUseEchoSuppression(true)`，除非您确信用户使用耳机来回放声音，或者使用除扬声器以外的某种设备。

以下代码说明了如何将来自本地麦克风的音频传送到本地系统扬声器：

```
var mic:Microphone = Microphone.getMicrophone();
mic.setUseEchoSuppression(true);
mic.setLoopBack(true);
```

## 更改麦克风音频

应用程序可以使用两种方法更改来自麦克风的音频数据。第一，它可以更改输入声音的增益，这会有效地将输入值乘以指定的数值以创建更大或更小的声音。`Microphone.gain` 属性接受介于 0 和 100 之间的数值（含 0 和 100）。值 50 相当于乘数 1，它指定正常音量。值 0 相当于乘数 0，它可有效地将输入音频静音。大于 50 的值指定的音量高于正常音量。

应用程序也可以更改输入音频的采样率。较高的采样率可提高声音品质，但它们也会创建更密集的数据流（使用更多的资源进行传输和存储）。`Microphone.rate` 属性表示以千赫 (kHz) 为单位测量的音频采样率。默认采样率是 8 kHz。如果麦克风支持较高的采样率，您可以将 `Microphone.rate` 属性设置为高于 8 kHz 的值。例如，如果将 `Microphone.rate` 属性设置为值 11，则会将采样率设置为 11 kHz ；如果将其设置为 22，则会将采样率设置为 22 kHz，依此类推。

## 检测麦克风活动

为节省带宽和处理资源， Flash Player 将尝试检测何时麦克风不传输声音。当麦克风的活动级别处于静音级别阈值以下一段时间后， Flash Player 将停止传输音频输入，并调度一个简单的 ActivityEvent。

Microphone 类的以下三个属性用于监视和控制活动检测：

- `activityLevel` 只读属性指示麦克风检测的音量，范围从 0 到 100。
- `silenceLevel` 属性指定激活麦克风并调度 `ActivityEvent.ACTIVITY` 事件所需的音量。`silenceLevel` 属性也使用从 0 到 100 的范围，默认值为 10。
- `silenceTimeout` 属性描述活动级别处于静音级别以下多长时间 （以毫秒为单位）后，才会调度 `ActivityEvent.ACTIVITY` 事件以指示麦克风现在处于静音状态。`silenceTimeout` 默认值是 2000。

`Microphone.silenceLevel` 属性和 `Microphone.silenceTimeout` 属性都是只读的，但可以使用 `Microphone.setSilenceLevel()` 方法来更改它们的值。

在某些情况下，在检测到新活动时激活麦克风的过程可能会导致短暂的延迟。通过将麦克风始终保持活动状态，可以消除此类激活延迟。应用程序可以调用 `Microphone.setSilenceLevel()` 方法并将 `silenceLevel` 参数设置为零，以通知 Flash Player 将麦克风保持活动状态并持续收集音频数据，即使未检测到任何声音也是如此。反之，如果将 `silenceLevel` 参数设置为 100，则可以完全禁止激活麦克风。

以下示例显示了有关麦克风的信息，并报告 Microphone 对象调度的活动事件和状态事件：

```
import flash,events.ActivityEvent;
import flash,events.StatusEvent;
import flash.media.Microphone;

var deviceArray:Array = Microphone.names;
trace("Available sound input devices:");
for (var i:int = 0; i < deviceArray.length; i++)
{
  trace("    " + deviceArray[i]);
}

var mic:Microphone = Microphone.getMicrophone();
mic.gain = 60;
mic.rate = 11;
mic.setUseEchoSuppression(true);
mic.setLoopBack(true);
mic.setSilenceLevel(5, 1000);

mic.addEventListener(ActivityEvent.ACTIVITY, this.onMicActivity);
mic.addEventListener(StatusEvent.STATUS, this.onMicStatus);

var micDetails:String = "Sound input device name: " + mic.name + '\n';
```

```
micDetails += "Gain: " + mic.gain + '\n';
micDetails += "Rate: " + mic.rate + " kHz" + '\n';
micDetails += "Muted: " + mic.muted + '\n';
micDetails += "Silence level: " + mic.silenceLevel + '\n';
micDetails += "Silence timeout: " + mic.silenceTimeout + '\n';
micDetails += "Echo suppression: " + mic.useEchoSuppression + '\n';
trace(micDetails);

function onMicActivity(event:ActivityEvent):void
{
  trace("activating=" + event.activating + ", activityLevel=" +
    mic.activityLevel);
}

function onMicStatus(event:StatusEvent):void
{
  trace("status: level=" + event.level + ", code=" + event.code);
}
```

在运行上面的示例时，对着系统麦克风说话或发出噪音，并观察所生成的、显示在控制台或调试窗口中的 trace 语句。

## 向媒体服务器发送音频以及从中接收音频

将 ActionScript 与 Flash Media Server 等流媒体服务器配合使用时，可以使用额外的音频功能。

特别地，应用程序可以将 Microphone 对象附加到 NetStream 对象上，并将数据直接从用户麦克风传输到服务器。也可以将音频数据从服务器流式传输到 Flash 或 Flex 应用程序，并将其作为 MovieClip 的一部分或使用 Video 对象进行回放。

有关详细信息，请参阅 http://livedocs.macromedia.com 上提供的在线 Flash Media Server 文档。

# 示例：Podcast Player

播客是通过 Internet 以按需方式或订阅方式分发的声音文件。播客通常是作为系列的一部分发布的，此系列也称为播客频道。由于播客节目的持续时间从一分钟到数小时不等，因此，通常在播放的同时对其进行流式传输。播客节目 （也称为项目）通常是以 mp3 文件格式提供的。视频播客也非常受欢迎，但此范例应用程序仅播放使用 mp3 文件的音频播客。

此示例并不是一个功能完备的播客聚合器应用程序。例如，它不能管理对特定播客的订阅，或在下次运行应用程序时记住用户已收听的播客。它可用作功能更完备的播客聚合器的起点。

Podcast Player 示例说明了以下 ActionScript 编程方法：

- 读取外部 RSS 新闻频道并分析其 XML 内容
- 创建 SoundFacade 类以简化加载和回放声音文件的过程
- 显示声音回放进度
- 暂停和恢复声音回放

要获取该范例的应用程序文件，请访问 www.adobe.com/go/learn_programmingAS3samples_flash_cn。 Podcast Player 应用程序文件位于文件夹 Samples/PodcastPlayer 中。该应用程序包含以下文件：

| 文件 | 描述 |
|---|---|
| PodcastPlayer.mxml<br>或<br>PodcastPlayer.fla | 适用于 Flex (MXML) 或 Flash (FLA) 的应用程序的用户界面。 |
| RSSBase.as | 为 RSSChannel 类和 RSSItem 类提供公共属性和方法的基类。 |
| RSSChannel.as | 保存 RSS 频道的相关数据的 ActionScript 类。 |
| RSSItem.as | 保存 RSS 项目的相关数据的 ActionScript 类。 |
| SoundFacade.as | 应用程序的主 ActionScript 类。它封装 Sound 类和 SoundChannel 类的方法和事件，并添加对回放暂停和恢复的支持。 |
| URLService.as | 从远程 URL 检索数据的 ActionScript 类。 |
| playerconfig.xml | 这是一个 XML 文件，其中包含表示播客频道的 RSS 新闻频道列表。 |

# 读取播客频道的 RSS 数据

Podcast Player 应用程序先读取一些播客频道及其节目的相关信息：

1. 首先，应用程序读取包含播客频道列表的 XML 配置文件，并向用户显示该频道列表。

2. 当用户选择其中的一个播客频道时，它将读取该频道的 RSS 新闻频道并显示频道节目列表。

此示例使用 URLLoader 实用程序类，从远程位置或本地文件检索基于文本的数据。Podcast Player 先创建一个 URLLoader 对象，以便从 playerconfig.xml 文件中获取采用 XML 格式的 RSS 新闻频道列表。接下来，当用户从列表中选择特定新闻频道时，将创建一个新的 URLLoader 对象以从该新闻频道的 URL 中读取 RSS 数据。

## 使用 SoundFacade 类简化声音加载和回放

ActionScript 3.0 声音体系结构功能强大，但非常复杂。如果应用程序只需要基本的声音加载和回放功能，可使用通过提供一组更简单的方法调用和事件来隐藏某些复杂性的类。在软件设计模式领域中，这样的类称为 “外观”。

SoundFacade 类表示用于执行以下任务的单个接口：

- 使用 Sound 对象、 SoundLoaderContext 对象以及 SoundMixer 类来加载声音文件
- 使用 Sound 对象和 SoundChannel 对象来播放声音文件
- 调度回放进度事件
- 使用 Sound 对象和 SoundChannel 对象来暂停和恢复声音回放

SoundFacade 类尝试以更简化的方式提供 ActionScript Sound 类的大部分功能。

以下代码显示了类声明、类属性以及 `SoundFacade()` 构造函数方法：

```
public class SoundFacade extends EventDispatcher
{
    public var s:Sound;
    public var sc:SoundChannel;
    public var url:String;
    public var bufferTime:int = 1000;

    public var isLoaded:Boolean = false;
    public var isReadyToPlay:Boolean = false;
    public var isPlaying:Boolean = false;
    public var isStreaming:Boolean = true;
    public var autoLoad:Boolean = true;
    public var autoPlay:Boolean = true;

    public var pausePosition:int = 0;

    public static const PLAY_PROGRESS:String = "playProgress";
    public var progressInterval:int = 1000;
    public var playTimer:Timer;

    public function SoundFacade(soundUrl:String, autoLoad:Boolean = true,
                        autoPlay:Boolean = true, streaming:Boolean = true,
                        bufferTime:int = -1):void
    {
        this.url = soundUrl;

        // 设置决定此对象行为的布尔值
        this.autoLoad = autoLoad;
        this.autoPlay = autoPlay;
        this.isStreaming = streaming;

        // 采用默认全局 bufferTime 值
        if (bufferTime < 0)
```

```
    {
      bufferTime = SoundMixer.bufferTime;
    }

    // 将缓冲时间保持在合理的范围内：介于 0 和 30 秒之间
    this.bufferTime = Math.min(Math.max(0, bufferTime), 30000);

    if (autoLoad)
    {
      load();
    }
}
```

SoundFacade 类扩展了 EventDispatcher 类，以使其能够调度自己的事件。类代码先声明 Sound 对象和 SoundChannel 对象的属性。该类还会存储声音文件 URL 的值以及对声音进行流式传输时使用的 bufferTime 属性。此外，它还接受某些影响加载和回放行为的布尔参数值：

- autoLoad 参数通知对象，应在创建此对象后立即启动声音加载。
- autoPlay 参数指示在加载了足够多的声音数据后应立即启动声音播放。如果这是声音流，在加载了足够多的数据（由 bufferTime 属性指定）后将立即开始回放。
- streaming 参数指示可以在加载完成之前开始播放此声音文件。

bufferTime 参数的默认值为 -1。如果构造函数方法在 bufferTime 参数中检测到负值，它会将 bufferTime 属性设置为 SoundMixer.bufferTime 的值。这样，应用程序便可根据需要默认使用全局 SoundMixer.bufferTime 值。

如果将 autoLoad 参数设置为 true，构造函数方法将立即调用以下 load() 方法来开始加载声音文件：

```
public function load():void
{
  if (this.isPlaying)
  {
    this.stop();
    this.s.close();
  }
  this.isLoaded = false;

  this.s = new Sound();

  this.s.addEventListener(ProgressEvent.PROGRESS, onLoadProgress);
  this.s.addEventListener(Event.OPEN, onLoadOpen);
  this.s.addEventListener(Event.COMPLETE, onLoadComplete);
  this.s.addEventListener(Event.ID3, onID3);
  this.s.addEventListener(IOErrorEvent.IO_ERROR, onIOError);
  this.s.addEventListener(SecurityErrorEvent.SECURITY_ERROR, onIOError);

  var req:URLRequest = new URLRequest(this.url);
```

```
  var context:SoundLoaderContext = new SoundLoaderContext(this.bufferTime,
                                                          true);
  this.s.load(req, this.slc);
}
```

load() 方法创建一个新的 Sound 对象，然后为所有重要的声音事件添加侦听器。接下来，它通知 Sound 对象使用 SoundLoaderContext 对象传入 bufferTime 值以加载声音文件。

由于可以更改 url 属性，因此，可以使用 SoundFacade 实例来连续播放不同的声音文件：只需更改 url 属性并调用 load() 方法，即可加载新的声音文件。

以下三个事件侦听器方法说明了 SoundFacade 对象如何跟踪加载进度并确定何时开始播放声音：

```
public function onLoadOpen(event:Event):void
{
  if (this.isStreaming)
  {
    this.isReadyToPlay = true;
    if (autoPlay)
    {
      this.play();
    }
  }
  this.dispatchEvent(event.clone());
}

public function onLoadProgress(event:ProgressEvent):void
{
  this.dispatchEvent(event.clone());
}

public function onLoadComplete(event:Event):void
{
  this.isReadyToPlay = true;
  this.isLoaded = true;
  this.dispatchEvent(evt.clone());

  if (autoPlay && !isPlaying)
  {
    play();
  }
}
```

在开始加载声音时，将执行 onLoadOpen() 方法。如果可以使用流模式播放声音，onLoadComplete() 方法会立即将 isReadyToPlay 标志设置为 true。isReadyToPlay 标志确定应用程序能否开始播放声音，这可能为了响应用户动作，如单击“播放”按钮。SoundChannel 类管理声音数据的缓冲，因此在调用 play() 方法之前，不需要显式地检查是否加载了足够多的数据。

在加载过程中，将定期执行 onLoadProgress() 方法。它仅调度其 ProgressEvent 对象的克隆，该对象由使用此 SoundFacade 对象的代码使用。

当完全加载了声音数据后，将执行 onLoadComplete() 方法，以便为非声音流调用 play() 方法（如果需要）。下面显示了 play() 方法本身。

```
public function play(pos:int = 0):void
{
  if (!this.isPlaying)
  {
    if (this.isReadyToPlay)
    {
      this.sc = this.s.play(pos);
      this.sc.addEventListener(Event.SOUND_COMPLETE, onPlayComplete);
      this.isPlaying = true;

      this.playTimer = new Timer(this.progressInterval);
      this.playTimer.addEventListener(TimerEvent.TIMER, onPlayTimer);
      this.playTimer.start();
    }
  }
}
```

如果已准备好播放声音，play() 方法将调用 Sound.play() 方法。生成的 SoundChannel 对象存储在 sc 属性中。play() 方法随后创建一个 Timer 对象，该对象用于按固定间隔调度回放进度事件。

## 显示回放进度

创建 Timer 对象以实现回放监视是一个很复杂的操作，您只需对其编码一次。通过在可重用的类（如 SoundFacade 类）中封装此 Timer 逻辑，应用程序可以在加载和播放声音时侦听相同类型的进度事件。

由 SoundFacade.play() 方法创建的 Timer 对象每秒调度一个 TimerEvent 实例。每当新的 TimerEvent 到达时，就会执行以下 onPlayTimer() 方法：

```
public function onPlayTimer(event:TimerEvent):void
{
  var estimatedLength:int =
    Math.ceil(this.s.length / (this.s.bytesLoaded / this.s.bytesTotal));
  var progEvent:ProgressEvent =
    new ProgressEvent(PLAY_PROGRESS, false, false, this.sc.position,
  estimatedLength);
  this.dispatchEvent(progEvent);
}
```

onPlayTimer() 方法可实现第 523 页的 “监视回放”一节中描述的大小估计方法。然后，它创建一个事件类型为 SoundFacade.PLAY_PROGRESS 的新 ProgressEvent 实例，并将 bytesLoaded 属性设置 SoundChannel 对象的当前位置，而将 bytesTotal 属性设置为估计的声音数据长度。

## 暂停和恢复回放

以前显示的 SoundFacade.play() 方法接受 pos 参数，该参数与声音数据中的起始位置相对应。如果 pos 值为零，则从开头开始播放声音。

SoundFacade.stop() 方法也接受 pos 参数，如下所示：

```
public function stop(pos:int = 0):void
{
    if (this.isPlaying)
    {
        this.pausePosition = pos;
        this.sc.stop();
        this.playTimer.stop();
        this.isPlaying = false;
    }
}
```

每当调用 SoundFacade.stop() 方法时，它都会设置 pausePosition 属性，以便当用户要恢复回放相同的声音时应用程序知道将播放头放置在什么位置。

下面显示的 SoundFacade.pause() 和 SoundFacade.resume() 方法分别调用 SoundFacade.stop() 和 SoundFacade.play() 方法，以便每次传递 pos 参数值。

```
public function pause():void
{
    stop(this.sc.position);
}

public function resume():void
{
    play(this.pausePosition);
}
```

pause() 方法将当前的 SoundChannel.position 值传递给 play() 方法，后者将该值存储在 pausePosition 属性中。 resume() 方法通过使用 pausePosition 值作为起点再次开始播放相同的声音。

## 扩展 Podcast Player 示例

此示例呈现了一个框架 Podcast Player，用于说明如何使用可重用的 SoundFacade 类。您可以添加其它功能以改进此应用程序的用途，其中包括以下功能：

- 将新闻频道列表和有关每个节目的使用信息存储在 SharedObject 实例中，下次用户运行应用程序时便可以使用它们。
- 允许用户将其自己的 RSS 新闻频道添加到播客频道列表中。
- 当用户停止或离开节目时，记住播放头的位置，以便下次用户运行应用程序时能够从该位置重新开始播放。

- 下载节目的 mp3 文件，以便用户在没有连接到 Internet 时可以脱机收听。
- 添加订阅功能，以便定期检查播客频道中的新节目并自动更新节目列表。
- 使用来自播客托管服务（如 Odeo.com）的 API 添加播客搜索和浏览功能。

第 21 章

# 捕获用户输入

21

本章介绍了如何使用 ActionScript 3.0 来创建交互性以响应用户活动。它讨论了键盘和鼠标事件，然后继续介绍更高级的主题，其中包括上下文菜单自定义和焦点管理。本章最后介绍了 WordSearch，这是一个响应鼠标输入的应用程序示例。

请注意，本章假定您已熟悉了 ActionScript 3.0 事件模型。有关详细信息，请参阅第 267 页的第 10 章 “处理事件”。

## 目录



# 用户输入基础知识

## 捕获用户输入简介

用户交互（无论是通过键盘、鼠标、摄像头还是这些设备的组合）是交互性的基础。在 ActionScript 3.0 中，识别和响应用户交互主要涉及事件侦听。

InteractiveObject 类是 DisplayObject 类的一个子类，它提供了处理用户交互所需的事件和功能的通用结构。您无法直接创建 InteractiveObject 类的实例。而是由显示对象（如 SimpleButton、Sprite、TextField 和各种 Flash 和 Flex 组件）从此类中继承其用户交互模型，因而它们使用同一个通用结构。这意味着，您为处理从 InteractiveObject 派生的一个对象中的用户交互而编写的代码以及学会的方法适用于所有其它对象。

本章介绍了以下典型的用户交互任务：

- 捕获应用程序范围内的键盘输入
- 捕获特定显示对象的键盘输入
- 捕获应用程序范围内的鼠标动作

- 捕获特定显示对象的鼠标输入
- 创建拖放交互性
- 创建自定义鼠标光标 （鼠标指针）
- 将新行为添加到上下文菜单中
- 管理焦点

## 重要概念和术语

在继续阅读本章内容之前，一定要先熟悉以下重要用户交互术语：

- 字符代码 (Character code)：表示当前字符集中的字符 （与在键盘上所按的键关联）的数字代码。例如，尽管 “D”和 “d”是由美国英语键盘上的相同键创建的，但它们具有不同的字符代码。
- 上下文菜单 (Context menu)：当用户右键单击或使用特定键盘 — 鼠标组合时显示的菜单。上下文菜单命令通常直接应用于已单击的内容。例如，某个图像的上下文菜单可能包含用于在单独窗口中显示该图像以及下载该图像的命令。
- 焦点 (Focus)：指示选定元素是活动元素，并且它是键盘或鼠标交互的目标。
- 键控代码 (Key code)：对应于键盘上的实际键的数字代码。

## 完成本章中的示例

学习本章的过程中，您可能想要自己动手测试一些范例代码清单。由于本章介绍的是在 ActionScript 中处理用户输入，因此本章中的几乎所有代码清单都涉及操作某一类型的显示对象 — 通常是文本字段或任何 InteractiveObject 子类。对于这些示例而言，显示对象可以是已经创建并放置在 Adobe Flash CS3 Professional 中的舞台上的显示对象，也可以是使用 ActionScript 创建的显示对象。测试范例涉及在 Flash Player 中查看结果，并与范例交互以查看代码的效果。

**要测试本章中的代码清单，请执行以下操作：**

1. 创建一个空的 Flash 文档。
2. 在时间轴上选择一个关键帧。
3. 打开 “动作”面板，将代码清单复制到 “脚本”窗格中。
4. 在舞台上创建一个实例：
   - 如果代码引用一个文本字段，请使用 “文本”工具在舞台上创建一个动态文本字段。
   - 否则，在舞台上创建一个按钮或影片剪辑元件实例。
5. 选择该文本字段、按钮或影片剪辑，并在 “属性”检查器中为它指定一个实例名。该名称应与范例代码中的显示对象的名称匹配。例如，如果代码操作一个名为 `myDisplayObject` 的对象，则将您的舞台对象也命名为 `myDisplayObject`。

6. 使用 “控制” > “测试影片”运行程序。

在屏幕上，将按照代码中的指定操作显示对象。

# 捕获键盘输入

从 InteractiveObject 类继承交互模型的显示对象可以使用事件侦听器来响应键盘事件。例如，您可以将事件侦听器放在舞台上以侦听并响应键盘输入。在以下代码中，事件侦听器捕获一个按键，并显示键名和键控代码属性：

```
function reportKeyDown(event:KeyboardEvent):void
{
  trace("Key Pressed: " + String.fromCharCode(event.charCode) +
    " (character code: " + event.charCode + ")");
}
stage.addEventListener(KeyboardEvent.KEY_DOWN, reportKeyDown);
```

有些键（如 Ctrl 键）虽然没有字型表示形式，也能生成事件。

在上面的代码示例中，键盘事件侦听器捕获了整个舞台的键盘输入。也可以为舞台上的特定显示对象编写事件侦听器；当对象具有焦点时将触发该事件侦听器。

在以下示例中，仅当用户在 TextField 实例内键入内容时，才会在 “输出”面板中反映键击。按下 Shift 键可暂时将 TextField 的边框颜色更改为红色。

此代码假定舞台上有一个名为 `tf` 的 TextField 实例。

```
tf.border = true;
tf.type = "input";
tf.addEventListener(KeyboardEvent.KEY_DOWN,reportKeyDown);
tf.addEventListener(KeyboardEvent.KEY_UP,reportKeyUp);

function reportKeyDown(event:KeyboardEvent):void
{
  trace("Key Pressed: " + String.fromCharCode(event.charCode) +
    " (key code: " + event.keyCode + " character code: "
    + event.charCode + ")");
  if (event.keyCode == Keyboard.SHIFT) tf.borderColor = 0xFF0000;
}

function reportKeyUp(event:KeyboardEvent):void
{
  trace("Key Released: " + String.fromCharCode(event.charCode) +
    " (key code: " + event.keyCode + " character code: " +
    event.charCode + ")");
  if (event.keyCode == Keyboard.SHIFT)
  {
    tf.borderColor = 0x000000;
  }
}
```

TextField 类还会报告 textInput 事件，当用户输入文本时，您可以侦听该事件。有关详细信息，请参阅第 447 页的 “捕获文本输入”。

# 了解键控代码和字符代码

您可以访问键盘事件的 keyCode 和 charCode 属性，以确定按下了哪个键，然后触发其它动作。keyCode 属性为数值，与键盘上的某个键的值相对应。charCode 属性是该键在当前字符集中的数值。（默认字符集是 UTF-8，它支持 ASCII。）

键控代码值与字符值之间的主要区别是键控代码值表示键盘上的特定键（数字小键盘上的 1 与最上面一排键中的 1 不同，但生成 “1” 的键与生成 “!” 的键是相同的），字符值表示特定字符 （R 与 r 字符是不同的）。



有关各个键与其在 ASCII 中的字符代码值之间的映射，请参阅第 593 页的附录 C “键盘键和键控代码值”。

键与其键控代码之间的映射取决于设备和操作系统。因此，不应使用键映射来触发动作，而应使用 Keyboard 类提供的预定义常量值来引用相应的 keyCode 属性。例如，不要使用 Shift 的键映射，而应使用 Keyboard.SHIFT 常量 （如上面的代码范例中所示）。

# 了解 KeyboardEvent 的优先顺序

与其它事件一样，键盘事件序列是由显示对象层次结构决定的，而不是由在代码中分配 addEventListener() 方法的顺序决定的。

例如，假定您将名为 tf 的文本字段放在名为 container 的影片剪辑内，并在这两个实例中添加键盘事件的事件侦听器，如下例所示：

```
container.addEventListener(KeyboardEvent.KEY_DOWN,reportKeyDown);
container.tf.border = true;
container.tf.type = "input";
container.tf.addEventListener(KeyboardEvent.KEY_DOWN,reportKeyDown);

function reportKeyDown(event:KeyboardEvent):void
{
  trace(event.currentTarget.name + " hears key press: " +
    String.fromCharCode(event.charCode) + " (key code: " +
    event.keyCode + " character code: " + event.charCode + ")");
}
```

由于文本字段及其父容器中均包含侦听器，因此，将为 TextField 内的每次键击调用两次 reportKeyDown() 函数。请注意，对于每次按键操作，文本字段在 container 影片剪辑调度事件之前调度事件。

操作系统和 Web 浏览器在 Adobe Flash Player 之前处理键盘事件。例如，在 Microsoft Internet Explorer 中按 Ctrl+W 将先关闭浏览器窗口，然后再由包含的任何 SWF 文件调度键盘事件。

# 捕获鼠标输入

鼠标单击将创建鼠标事件，这些事件可用来触发交互式功能。您可以将事件侦听器添加到舞台上以侦听在 SWF 文件中任何位置发生的鼠标事件。也可以将事件侦听器添加到舞台上从 InteractiveObject 进行继承的对象（例如，Sprite 或 MovieClip）中；单击该对象时将触发这些侦听器。

与键盘事件一样，鼠标事件也会冒泡。在下面的示例中，由于 `square` 是 Stage 的子级，因此，单击正方形时，将从 Sprite `square` 和 Stage 对象中调度该事件：

```
var square:Sprite = new Sprite();
square.graphics.beginFill(0xFF0000);
square.graphics.drawRect(0,0,100,100);
square.graphics.endFill();
square.addEventListener(MouseEvent.CLICK, reportClick);
square.x =
square.y = 50;
addChild(square);

stage.addEventListener(MouseEvent.CLICK, reportClick);

function reportClick(event:MouseEvent):void
{
  trace(event.currentTarget.toString() +
    " dispatches MouseEvent.Local coords [" +
    event.localX + "," + event.localY + "] Stage coords [" +
    event.stageX + "," + event.stageY + "]");
}
```

请注意，在上面的示例中，鼠标事件包含有关单击的位置信息。`localX` 和 `localY` 属性包含显示链中最低级别的子级上的单击位置。例如，单击 `square` 左上角时将报告本地坐标 [0,0]，因为它是 `square` 的注册点。或者，`stageX` 和 `stageY` 属性是指单击位置在舞台上的全局坐标。同一单击报告这些坐标为 [50,50]，因为 `square` 已移到这些坐标上。取决于响应用户交互的方式，这两种坐标对可能是非常有用的。

MouseEvent 对象还包含 `altKey`、`ctrlKey` 和 `shiftKey` 布尔属性。可以使用这些属性来检查在鼠标单击时是否还按下了 Alt、 Ctrl 或 Shift 键。

## 创建拖放功能

通过使用拖放功能，用户可以在按鼠标左键的同时选择对象，将该对象移到屏幕上的新位置，然后松开鼠标左键以将其放在新位置上。下面的代码显示了一个这样的示例：

```
import flash.display.Sprite;
import flash.events.MouseEvent;

var circle:Sprite = new Sprite();
circle.graphics.beginFill(0xFFCC00);
circle.graphics.drawCircle(0, 0, 40);

var target1:Sprite = new Sprite();
target1.graphics.beginFill(0xCCFF00);
target1.graphics.drawRect(0, 0, 100, 100);
target1.name = "target1";

var target2:Sprite = new Sprite();
target2.graphics.beginFill(0xCCFF00);
target2.graphics.drawRect(0, 200, 100, 100);
target2.name = "target2";

addChild(target1);
addChild(target2);
addChild(circle);

circle.addEventListener(MouseEvent.MOUSE_DOWN, mouseDown)

function mouseDown(event:MouseEvent):void
{
  circle.startDrag();
}
circle.addEventListener(MouseEvent.MOUSE_UP, mouseReleased);

function mouseReleased(event:MouseEvent):void
{
  circle.stopDrag();
  trace(circle.dropTarget.name);
}
```

有关更多详细信息，请参阅第 341 页的 “创建拖放交互组件”。

## 自定义鼠标光标

可以将鼠标光标（鼠标指针）隐藏或交换为舞台上的任何显示对象。要隐藏鼠标光标，请调用 Mouse.hide() 方法。可通过以下方式来自定义光标：调用 Mouse.hide()，侦听舞台上是否发生 MouseEvent.MOUSE_MOVE 事件，以及将显示对象（自定义光标）的坐标设置为事件的 stageX 和 stageY 属性。下面的示例说明了此任务的基本执行过程：

```
var cursor:Sprite = new Sprite();
cursor.graphics.beginFill(0x000000);
cursor.graphics.drawCircle(0,0,20);
cursor.graphics.endFill();
addChild(cursor);

stage.addEventListener(MouseEvent.MOUSE_MOVE,redrawCursor);
Mouse.hide();

function redrawCursor(event:MouseEvent):void
{
  cursor.x = event.stageX;
  cursor.y = event.stageY;
}
```

## 自定义上下文菜单

从 InteractiveObject 类进行继承的每个对象可以具有唯一的上下文菜单，用户在 SWF 文件内右键单击时将显示该菜单。默认情况下，菜单中包含几个命令，其中包括“前进”、“后退”、“打印”、“品质”和“缩放”。

除了“设置”和“关于”命令外，您可以从菜单中删除所有其它默认命令。如果将 Stage 属性 showDefaultContextMenu 设置为 false，则会从上下文菜单中删除这些命令。

要为特定显示对象创建自定义的上下文菜单，请创建 ContextMenu 类的一个新实例，调用 hideBuiltInItems() 方法，并将该实例分配给该 DisplayObject 实例的 contextMenu 属性。下面的示例为一个动态绘制的正方形提供了一个上下文菜单命令，用于将其更改为随机颜色：

```
var square:Sprite = new Sprite();
square.graphics.beginFill(0x000000);
square.graphics.drawRect(0,0,100,100);
square.graphics.endFill();
square.x =
square.y = 10;
addChild(square);

var menuItem:ContextMenuItem = new ContextMenuItem("Change Color");
menuItem.addEventListener(ContextMenuEvent.MENU_ITEM_SELECT,changeColor);
var customContextMenu:ContextMenu = new ContextMenu();
customContextMenu.hideBuiltInItems();
```

```
customContextMenu.customItems.push(menuItem);
square.contextMenu = customContextMenu;

function changeColor(event:ContextMenuEvent):void
{
  square.transform.colorTransform = getRandomColor();
}
function getRandomColor():ColorTransform
{
  return new ColorTransform(Math.random(), Math.random(),
    Math.random(),1,(Math.random() * 512) - 255,
    (Math.random() * 512) -255, (Math.random() * 512) - 255, 0);
}
```

## 管理焦点

交互式对象可以按编程方式或通过用户动作来获得焦点。在这两种情况下，设置焦点会将对象的 focus 属性更改为 true。另外，如果将 tabEnabled 属性设置为 true，用户可通过按 Tab 将焦点从一个对象传递到另一个对象。请注意，默认情况下，tabEnabled 值为 false，但以下情况除外：

- 对于 SimpleButton 对象，该值为 true。
- 对于输入文本字段，该值为 true。
- 对于 buttonMode 设置为 true 的 Sprite 或 MovieClip 对象，该值为 true。

在上述各种情况下，都可以为 FocusEvent.FOCUS_IN 或 FocusEvent.FOCUS_OUT 添加侦听器，以便在焦点更改时提供其它行为。这对文本字段和表单尤其有用，但也可以用于 sprite、影片剪辑或从 InteractiveObject 类进行继承的任何对象。下面的示例说明了如何使用 Tab 键启用焦点循环切换，以及如何响应后续的焦点事件。在本例中，每个正方形在收到焦点时将改变颜色。

注意：Flash 创作工具使用键盘快捷键来管理焦点；因此，要正确模拟焦点管理，应在浏览器中测试 SWF 文件，而不是在 Flash 中进行测试。

```
var rows:uint = 10;
var cols:uint = 10;
var rowSpacing:uint = 25;
var colSpacing:uint = 25;
var i:uint;
var j:uint;
for (i = 0; i < rows; i++)
{
  for (j = 0; j < cols; j++)
  {
    createSquare(j * colSpacing, i * rowSpacing, (i * cols) + j);
  }
}
```

```
function createSquare(startX:Number, startY:Number, tabNumber:uint):void
{
  var square:Sprite = new Sprite();
  square.graphics.beginFill(0x000000);
  square.graphics.drawRect(0, 0, colSpacing, rowSpacing);
  square.graphics.endFill();
  square.x = startX;
  square.y = startY;
  square.tabEnabled = true;
  square.tabIndex = tabNumber;
  square.addEventListener(FocusEvent.FOCUS_IN, changeColor);
  addChild(square);
}
function changeColor(event:FocusEvent):void
{
  e.target.transform.colorTransform = getRandomColor();
}
function getRandomColor():ColorTransform
{
  // 为红色、绿色和蓝色通道生成随机值。
  var red:Number = (Math.random() * 512) - 255;
  var green:Number = (Math.random() * 512) - 255;
  var blue:Number = (Math.random() * 512) - 255;

  // 使用随机颜色创建并返回 ColorTransform 对象。
  return new ColorTransform(1, 1, 1, 1, red, green, blue, 0);
}
```

# 示例：WordSearch

此示例通过处理鼠标事件来说明用户交互。用户在一个由随机字母组成的网格上构造尽可能多的词，拼写方法是：在网格中进行水平或垂直移动，但同一个字母只允许使用一次。此示例演示了 ActionScript 3.0 的下列功能：

- 动态构造组件网格
- 响应鼠标事件
- 根据用户交互维护分数

要获取该范例的应用程序文件，请访问
www.adobe.com/go/learn_programmingAS3samples_flash_cn。可以在 Samples/WordSearch 文件夹中找到 WordSearch 应用程序文件。该应用程序包含以下文件：

| 文件 | 说明 |
| --- | --- |
| WordSearch.as | 此类提供了应用程序的主要功能。 |
| WordSearch.fla | 适用于 Flash 的主应用程序文件。 |
| dictionary.txt | 此文件用于确定拼写的词能否得分以及拼写是否正确。 |

## 加载字典

要创建一个涉及查找词的游戏，您需要使用字典。本示例包含一个名为 dictionary.txt 的文本文件，该文件包含以回车符分隔的词列表。创建名为 `words` 的数组后，`loadDictionary()` 函数将请求加载该文件，成功加载后，此文件将变成一个很长的字符串。通过使用 `split()` 方法，在回车符 （字符代码 10）的每个实例处断开，可以将该字符串分析为词的数组。这种分析是在 `dictionaryLoaded()` 函数中实现的：

```
words = dictionaryText.split(String.fromCharCode(10));
```

## 创建用户界面

在存储这些词后，您就可以设置用户界面了。请创建两个 Button 实例：一个用于提交词，另一个用于清除当前拼写的词。在每种情况下，您必须通过侦听该按钮所广播的 `MouseEvent.CLICK` 事件并随后调用一个函数来响应用户输入。在 `setupUI()` 函数中，此代码在两个按钮上创建侦听器：

```
submitWordButton.addEventListener(MouseEvent.CLICK,submitWord);
clearWordButton.addEventListener(MouseEvent.CLICK,clearWord);
```

## 生成游戏板

游戏板是由随机字母组成的网格。在 `generateBoard()` 函数中，二维网格是通过将一个循环嵌套到另一个循环中创建的。第一个循环增加行数，第二个循环增加每行的总列数。由这些行和列创建的每个单元格包含一个按钮，它表示游戏板上的一个字母。

```
private function generateBoard(startX:Number,
    startY:Number,
    totalRows:Number,
    totalCols:Number,
    buttonSize:Number):void
{
    buttons = new Array();
    var colCounter:uint;
    var rowCounter:uint;
    for (rowCounter = 0; rowCounter < totalRows; rowCounter++)
    {
        for (colCounter = 0; colCounter < totalCols; colCounter++)
        {
            var b:Button = new Button();
            b.x = startX + (colCounter*buttonSize);
            b.y = startY + (rowCounter*buttonSize);
            b.addEventListener(MouseEvent.CLICK, letterClicked);
            b.label = getRandomLetter().toUpperCase();
            b.setSize(buttonSize,buttonSize);
            b.name = "buttonRow"+rowCounter+"Col"+colCounter;
            addChild(b);

            buttons.push(b);
        }
    }
}
```

虽然仅在一行中添加了 `MouseEvent.CLICK` 事件的侦听器，但由于该侦听器位于 `for` 循环中，因此会将其分配给每个 Button 实例。此外，还会为每个按钮分配一个从其行和列位置派生的名称，这为以后在代码中引用每个按钮的行和列提供了一种简便的方法。

## 通过用户输入来构造词

拼词规则是：选择垂直或水平相邻的字母，但每个字母只允许使用一次。每次单击时将生成一个鼠标事件，此时必须检查用户正在拼写的词，以确保该词正确地从以前单击的字母继续进行拼写。否则，将删除以前的词并开始拼写一个新词。此检查在 isLegalContinuation() 方法中执行。

```
private function isLegalContinuation(prevButton:Button,
  currButton:Button):Boolean
{
  var currButtonRow:Number = Number(currButton.name.charAt(currButton.name.
  indexOf("Row") + 3));
  var currButtonCol:Number =
  Number(currButton.name.charAt(currButton.name.indexOf("Col") + 3));
  var prevButtonRow:Number =
  Number(prevButton.name.charAt(prevButton.name.indexOf("Row") + 3));
  var prevButtonCol:Number =
  Number(prevButton.name.charAt(prevButton.name.indexOf("Col") + 3));

  return ((prevButtonCol == currButtonCol && Math.abs(prevButtonRow -
  currButtonRow) <= 1) ||
       (prevButtonRow == currButtonRow && Math.abs(prevButtonCol -
  currButtonCol) <= 1));
}
```

String 类的 charAt() 和 indexOf() 方法从当前单击的按钮和上次单击的按钮中检索相应的行和列。如果行或列未改变，或者行或列已改变但与上次单击位置的行或列之间相差不超过一行或一列，isLegalContinuation() 方法将返回 true。如果要更改游戏规则并允许斜向拼写，您可以删除对未改变的行或列的检查，最后一行将如下所示：

```
return (Math.abs(prevButtonRow - currButtonRow) <= 1) &&
  Math.abs(prevButtonCol - currButtonCol) <= 1));
```

## 检查词的提交

要完成游戏的代码，您需要提供检查词的提交和计算分数的机制。searchForWord() 方法包含这两项功能：

```
private function searchForWord(str:String):Number
{
  if (words && str)
  {
    var i:uint = 0
    for (i = 0; i < words.length; i++)
    {
      var thisWord:String = words[i];
      if (str == words[i])
      {
        return i;
```

```
      }
    }
    return -1;
  }
  else
  {
    trace("WARNING: cannot find words, or string supplied is null");
  }
  return -1;
}
```

此函数循环访问字典中所有的词。如果用户的词与字典中的词匹配，则返回该词在字典中的位置。如果该位置有效，submitWord() 方法随后将检查响应并更新分数。

## 自定义

类的开头是几个常量。可通过修改这些变量来修改此游戏。例如，您可以通过增大 TOTAL_TIME 变量的值来更改可用于玩该游戏的时间。还可以稍微增大 PERCENT_VOWELS 变量的值来增大找到词的可能性。

第 22 章

# 网络与通信

22

本章介绍如何使 SWF 文件与外部文件和其它 Adobe Flash Player 9 实例进行通信，还介绍如何从外部源加载数据，在 Java 服务器和 Flash Player 之间发送消息，以及使用 FileReference 和 FileReferenceList 类执行文件上载和下载。

## 目录

# 网络和通信基础知识

## 网络和通信简介

当构建更复杂的 ActionScript 应用程序时，通常需要与服务器端脚本进行通信，或者从外部 XML 文件或文本文件加载数据。 flash.net 包中包含用于通过 Internet 收发数据的类；例如，从远程 URL 加载内容、与其它 Flash Player 实例进行通信以及连接到远程网站。

而在 ActionScript 3.0，可以使用 URLLoader 和 URLRequest 类加载外部文件。可随后使用特定类来访问数据，具体取决于加载的数据类型。例如，如果将远程内容的格式设置为名称 - 值对，则可以使用 URLVariables 类来分析服务器结果。或者，如果使用 URLLoader 和 URLRequest 类加载的文件是远程 XML 文档，则可以使用 XML 类的构造函数、XMLDocument 类的构造函数或 `XMLDocument.parseXML()` 方法来分析 XML 文档。这样，您便可以简化 ActionScript 代码，因为无论是使用 URLVariables、XML 还是某个其它类来分析和处理远程数据，用于加载外部文件的代码都是相同的。

flash.net 包中还包含用于其它类型的远程通信的类。这些类包括 FileReference 类（用于将文件上载到服务器以及从服务器下载文件）、 Socket 和 XMLSocket 类（用于通过套接字连接直接与远程计算机进行通信）以及 NetConnection 和 NetStream 类（用于与特定于 Flash 的服务器资源（如 Flash Media Server 和 Flash Remoting 服务器）进行通信以及加载视频文件）。

最后， flash.net 包中包含用于用户本地计算机上通信的类。这些类包括 LocalConnection 类（用于在一台计算机上运行的两个或多个 SWF 文件之间的通信）和 SharedObject 类（用于将数据存储在用户的计算机上，并在以后返回到应用程序时检索这些数据）。

## 常见网络和通信任务

下表说明了需要从 ActionScript 中执行的与外部通信有关的常见任务；本章中对这些任务进行了介绍：

- 从外部文件或服务器脚本中加载数据
- 将数据发送到服务器脚本
- 与其它本地 SWF 文件通信
- 处理二进制套接字连接
- 与 XML 套接字通信
- 存储永久本地数据
- 将文件上载到服务器
- 将文件从服务器下载到用户计算机

## 重要概念和术语

以下参考列表包含将会在本章中遇到的重要术语：

- 外部数据 (External data)：此类数据以某些形式存储在 SWF 文件外部，并在需要时加载到 SWF 文件中。可以将此数据存储在直接加载的文件中，将其存储在数据库中，或者以其它形式进行存储，以便通过调用在服务器上运行的脚本或程序对其进行检索。
- URL 编码变量 (URL-encoded variable)：URL 编码格式提供了一种在单个文本字符串中表示多个变量（变量名和值对）的方法。各个变量采用 `name=value` 格式。每个变量（即每个名称 - 值对）之间用 "and" 符隔开，如下所示：`variable1=value1&variable2=value2`。这样，便可以将不限数量的变量作为一条消息进行发送。
- MIME 类型 (MIME type)：用于在 Internet 通信中标识给定文件类型的标准代码。任何给定文件类型都具有用于对其进行标识的特定代码。发送文件或消息时，计算机（如 Web 服务器或用户的 Flash Player 实例）将指定要发送的文件类型。
- HTTP：超文本传输协议，这是一种标准格式，用于传送通过 Internet 发送的网页和其它各种类型的内容。
- 请求方法 (Request method)：当程序（如 Flash Player）或 Web 浏览器将消息（称为 HTTP 请求）发送到 Web 服务器时，可以使用以下两种方法之一将发送的任何数据嵌入到请求中：这两种方法是两个 "请求方法"，即 GET 和 POST。在服务器端，接收请求的程序需要查看相应的请求部分以查找数据，因此，用于从 ActionScript 发送数据的请求方法应与用于在服务器上读取该数据的请求方法相匹配。
- 套接字连接 (Socket connection)：用于两台计算机之间的通信的永久连接。
- 上载 (Upload)：将文件发送到另一台计算机。
- 下载 (Download)：从另一台计算机检索文件。

## 完成本章中的示例

学习本章的过程中，您可能想要测试示例代码清单。本章中有几个代码清单加载外部数据或执行某些其它类型的通信；这些范例通常包括 `trace()` 函数调用，因此会在 "输出" 面板中显示示例的运行结果。其它示例则实际执行某一功能，例如将文件上载到服务器。测试这些示例将涉及与 SWF 交互并确认它们执行了所宣称的操作。

这些代码示例分为两类。一些示例列表是在假定代码在独立脚本（例如，附加到 Flash 文档中的关键帧的脚本）中的情况下编写的。要测试这些示例，请执行以下操作：

1. 创建一个新的 Flash 文档。
2. 选择时间轴的第 1 帧中的关键帧，并打开 "动作" 面板。
3. 将代码清单复制到 "脚本" 窗格中。
4. 从主菜单中，选择 "控制" > "测试影片" 以创建 SWF 文件并测试该示例。

其它示例代码清单编写为类的形式；示例类的预期功能是充当 Flash 文档的文档类。要测试这些示例，请执行以下操作：

1. 创建一个空的 Flash 文档并将它保存到您的计算机上。
2. 创建一个新的 ActionScript 文件，并将它保存到 Flash 文档所在的目录中。文件名应与代码清单中的类的名称一致。例如，如果代码清单定义一个名为 "UploadTest"的类，则将 ActionScript 文件保存为 "UploadTest.as"。
3. 将代码清单复制到 ActionScript 文件中并保存该文件。
4. 在 Flash 文档中，单击舞台或夹纸板的空白部分，以激活文档的 "属性"检查器。
5. 在"属性"检查器的"文档类"字段中，输入您从文本中复制的 ActionScript 类的名称。
6. 使用 "控制" > "测试影片"运行程序并测试该示例。

最后，本章中的一些示例涉及与在服务器上运行的程序进行交互。这些示例包括可用来创建测试示例所需的服务器程序的代码；您将需要在 Web 服务器计算机上设置适当的应用程序来测试这些示例。

# 处理外部数据

ActionScript 3.0 包含用于从外部源加载数据的机制。这些源可以是静态内容 （如文本文件），也可以是动态内容（如从数据库检索数据的 Web 脚本）。可以使用多种不同的方法来设置数据的格式，并且 ActionScript 提供了用于解码和访问数据的功能。也可以在检索数据的过程中将数据发送到外部服务器。

## 使用 URLLoader 类和 URLVariables 类

ActionScript 3.0 使用 URLLoader 和 URLVariables 类来加载外部数据。 URLLoader 类以文本、二进制数据或 URL 编码变量的形式从 URL 下载数据。 URLLoader 类用于下载文本文件、XML 或其它要用于数据驱动的动态 ActionScript 应用程序中的信息。URLLoader 类使用 ActionScript 3.0 高级事件处理模型，使用该模型可以侦听诸如 `complete`、`httpStatus`、`ioError`、`open`、`progress` 和 `securityError` 等事件。新事件处理模型在 ActionScript 2.0 的基础上大大改进了对 `LoadVars.onData`、 `LoadVars.onHTTPStatus` 和 `LoadVars.onLoad` 事件处理函数的支持，可以更高效地处理错误和事件。有关处理事件的详细信息，请参阅第 10 章 "处理事件"。

与早期版本 ActionScript 中的 XML 和 LoadVars 类非常相似， URLLoader URL 的数据在下载完成之前不可用。尽管如果文件加载速度太快，可能不会调度 `ProgressEvent.PROGRESS` 事件，但您可以通过侦听要调度的 `flash.events.ProgressEvent.PROGRESS` 事件来监视下载进度（已加载的字节数和总字节数）。成功下载文件后，将调度 `flash.events.Event.COMPLETE` 事件。加载的数据将从 UTF-8 或 UTF-16 编码被解码为字符串。



注意：如果没有为 `URLRequest.contentType` 设置值，则以 `application/x-www-form-urlencoded` 的形式发送值。

`URLLoader.load()` 方法（以及 URLLoader 类的构造函数，可选）使用一个参数，即 `request`，该参数是一个 URLRequest 实例。 URLRequest 实例包含单个 HTTP 请求的所有信息，如目标 URL、请求方法（`GET` 或 `POST`）、附加标头信息以及 MIME 类型（例如，当上载 XML 内容时）。

例如，要将 XML 数据包上载到服务器端脚本，您可以使用下面的 ActionScript 3.0 代码：

```
var secondsUTC:Number = new Date().time;
var dataXML:XML =
  <login>
    <time>{secondsUTC}</time>
    <username>Ernie</username>
    <password>guru</password>
  </login>;
var request:URLRequest = new URLRequest("http://www.yourdomain.com/
  login.cfm");
request.contentType = "text/xml";
request.data = dataXML.toXMLString();
request.method = URLRequestMethod.POST;
var loader:URLLoader = new URLLoader();
try
{
  loader.load(request);
}
catch (error:ArgumentError)
{
  trace("An ArgumentError has occurred.");
}
catch (error:SecurityError)
{
  trace("A SecurityError has occurred.");
}
```

上面的代码片段创建了一个名为 `dataXML` 的 XML 实例，其中包含要发送到服务器的 XML 数据包。接下来，将 URLRequest `contentType` 属性设置为 “`text/xml`”，将 URLRequest `data` 属性设置为 XML 数据包的内容（通过 `XML.toXMLString()` 方法将该内容转换为字符串）。最后，创建一个新的 URLLoader 实例，并使用 `URLLoader.load()` 方法将请求发送到远程脚本。

可以使用三种方式指定要在 URL 请求中传递的参数：

- 在 URLVariables 构造函数中
- 在 `URLVariables.decode()` 方法中
- 作为 URLVariables 对象本身中的特定属性

当定义 URLVariables 构造函数或 `URLVariables.decode()` 方法中的变量时，需要确保对 "and" 符进行 URL 编码，因为它具有特殊含义并作为分隔符使用。例如，由于与号作为参数的分隔符使用，当传递与号时，需要将与号从 `&` 更改为 `%26` 来对与号进行 URL 编码。

## 从外部文档加载数据

当使用 ActionScript 3.0 生成动态应用程序时，最好从外部文件或服务器端脚本加载数据。这样，您不必编辑或重新编译 ActionScript 文件，即可生成动态应用程序。例如，如果生成 "每日提示" 应用程序，您可以编写一个服务器端脚本，该脚本每天从数据库检索一次随机提示并将其保存到文本文件。然后， ActionScript 应用程序可以加载静态文本文件的内容，而不必每次查询数据库。

下面的片断创建 URLRequest 和 URLLoader 对象，用于加载外部文本文件 params.txt 的内容：

```
var request:URLRequest = new URLRequest("params.txt");
var loader:URLLoader = new URLLoader();
loader.load(request);
```

可以将上面的片断简化如下：

```
var loader:URLLoader = new URLLoader(new URLRequest("params.txt"));
```

默认情况下，如果您未定义请求方法， Flash Player 将使用 HTTP `GET` 方法加载内容。如果要使用 `POST` 方法发送数据，则需要使用静态常量 `URLRequestMethod.POST` 将 `request.method` 属性设置为 `POST`，如下面的代码所示：

```
var request:URLRequest = new URLRequest("sendfeedback.cfm");
request.method = URLRequestMethod.POST;
```

在运行时加载的外部文档 params.txt 包含以下数据：

```
monthNames=January,February,March,April,May,June,July,August,September,October,
   November,December&dayNames=Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,
   Saturday
```

该文件包含两个参数，即 `monthNames` 和 `dayNames`。每个参数包含一个逗号分隔列表，该列表被分解为字符串。可以使用 `String.split()` 方法将此列表拆分为数组。

| 提示 | 不要将保留字或语言构造作为外部数据文件中的变量名称，因为这样做会使代码的读取和调试变得更困难。 |
|---|---|

加载数据后，将调度 Event.COMPLETE 事件，现可以在 URLLoader 的 data 属性中使用外部文档的内容，如下面的代码所示：

```
private function completeHandler(event:Event):void
{
    var loader2:URLLoader = URLLoader(event.target);
    trace(loader2.data);
}
```

如果远程文档包含名称 - 值对，您可以通过传入加载文件的内容，使用 URLVariables 类来分析数据，如下所示：

```
private function completeHandler(event:Event):void
{
    var loader2:URLLoader = URLLoader(event.target);
    var variables:URLVariables = new URLVariables(loader2.data);
    trace(variables.dayNames);
}
```

外部文件中的每个名称 - 值对都创建为 URLVariables 对象中的一个属性。在上面的代码范例中，变量对象中的每个属性都被视为字符串。如果名称 - 值对是一个项目列表，您可以通过调用 String.split() 方法将字符串转换为数组，如下所示：

```
var dayNameArray:Array = variables.dayNames.split(",");
```

**提示** 如果从外部文本文件加载数值数据，则需要使用顶级函数（如 int()、uint() 或 Number()）将这些值转换为数值。

无需将远程文件的内容作为字符串加载和新建 URLVariables 对象，您可以将 URLLoader.dataFormat 属性设置为在 URLLoaderDataFormat 类中找到的静态属性之一。URLLoader.dataFormat 属性的三个可能值如下：

- URLLoaderDataFormat.BINARY — URLLoader.data 属性将包含 ByteArray 对象中存储的二进制数据。
- URLLoaderDataFormat.TEXT — URLLoader.data 属性将包含 String 对象中的文本。
- URLLoaderDataFormat.VARIABLES — URLLoader.data 属性将包含 URLVariables 对象中存储的 URL 编码的变量。

下面的代码说明了通过将 URLLoader.dataFormat 属性设置为 URLLoaderDataFormat.VARIABLES，您可以自动将加载的数据分解为 URLVariables 对象：

```
package
{
    import flash.display.Sprite;
    import flash.events.*;
    import flash.net.URLLoader;
    import flash.net.URLLoaderDataFormat;
    import flash.net.URLRequest;

    public class URLLoaderDataFormatExample extends Sprite
```

```
  {
    public function URLLoaderDataFormatExample()
    {
      var request:URLRequest = new URLRequest("http://www.[yourdomain].com/
  params.txt");
      var variables:URLLoader = new URLLoader();
      variables.dataFormat = URLLoaderDataFormat.VARIABLES;
      variables.addEventListener(Event.COMPLETE, completeHandler);
      try
      {
        variables.load(request);
      }
      catch (error:Error)
      {
        trace("Unable to load URL: " + error);
      }
    }
    private function completeHandler(event:Event):void
    {
        var loader:URLLoader = URLLoader(event.target);
        trace(loader.data.dayNames);
    }
  }
}
```

**注意** URLLoader.dataFormat 的默认值为 URLLoaderDataFormat.TEXT。

如下面的实例所示，从外部文件加载 XML 与加载 URLVariables 相同。可以创建 URLRequest 和 URLLoader 实例，然后使用它们下载远程 XML 文档。文件完全下载后，调度 Event.COMPLETE 事件，并将外部文件的内容转换为可使用 XML 方法和属性分析的 XML 实例。

```
package
{
  import flash.display.Sprite;
  import flash.errors.*;
  import flash.events.*;
  import flash.net.URLLoader;
  import flash.net.URLRequest;

  public class ExternalDocs extends Sprite
  {
    public function ExternalDocs()
    {
      var request:URLRequest = new URLRequest("http://www.[yourdomain].com/
  data.xml");
      var loader:URLLoader = new URLLoader();
      loader.addEventListener(Event.COMPLETE, completeHandler);
      try
```

```
        {
            loader.load(request);
        }
        catch (error:ArgumentError)
        {
            trace("An ArgumentError has occurred.");
        }
        catch (error:SecurityError)
        {
            trace("A SecurityError has occurred.");
        }
    }
    private function completeHandler(event:Event):void
    {
        var dataXML:XML = XML(event.target.data);
        trace(dataXML.toXMLString());
    }
  }
}
```

# 与外部脚本进行通信

除了加载外部数据文件，还可以使用 URLVariables 类将变量发送到服务器端脚本并处理服务器的响应。这是非常有用的，例如，如果您正在编写游戏，想要将用户的得分发送到服务器以计算是否应添加到高分列表中，也可以将用户的登录信息发送到服务器以进行验证。服务器端脚本可以处理用户名和密码，向数据库验证用户名和密码，然后返回用户提供的凭据是否有效的确认。

下面的片断创建一个名为 `variables` 的 URLVariables 对象，该对象创建称为 `name` 的新变量。接下来，创建一个 URLRequest 对象，该对象指定变量要发送到的服务器端脚本的 URL。然后，设置 URLRequest 对象的 `method` 属性，以便将变量作为 HTTP `POST` 请求发送。为了将 URLVariables 对象添加到 URL 请求，需要将 URLRequest 对象的 `data` 属性设置为早先创建的 URLVariables 对象。最后，创建 URLLoader 实例并调用 `URLLoader.load()` 方法，此方法用于启动该请求。

```
var variables:URLVariables = new URLVariables("name=Franklin");
var request:URLRequest = new URLRequest();
request.url = "http://www.[yourdomain].com/greeting.cfm";
request.method = URLRequestMethod.POST;
request.data = variables;
var loader:URLLoader = new URLLoader();
loader.dataFormat = URLLoaderDataFormat.VARIABLES;
loader.addEventListener(Event.COMPLETE, completeHandler);
try
{
    loader.load(request);
}
```

```
catch (error:Error)
{
  trace("Unable to load URL");
}

function completeHandler(event:Event):void
{
  trace(event.target.data.welcomeMessage);
}
```

下面的代码包含上面的示例中使用的 Adobe ColdFusion® greeting.cfm 文档的内容：

```
<cfif NOT IsDefined("Form.name") OR Len(Trim(Form.Name)) EQ 0>
  <cfset Form.Name = "Stranger" />
</cfif>
<cfoutput>welcomeMessage=#UrlEncodedFormat("Welcome, " & Form.name)#
</cfoutput>
```

# 连接到其它 Flash Player 实例

通过使用 LocalConnection 类，可以在不同的 Flash Player 实例（例如 HTML 容器、嵌入或独立播放器中的 SWF）之间进行通信。这样，您便可以构建在 Flash Player 实例之间共享数据（例如在 Web 浏览器中运行或嵌入在桌面应用程序中的 SWF 文件）的各种不同的应用程序。

## LocalConnection 类

LocalConnection 类用于开发 SWF 文件，这些文件无需使用 `fscommand()` 方法或 JavaScript 即可向其它 SWF 文件发送指令。 LocalConnection 对象只能在同一客户端计算机上运行的 SWF 文件间进行通信，但是它们可以在不同的应用程序中运行。例如，虽然放映文件维护本地信息，而基于浏览器的 SWF 进行的是远程连接，但在浏览器中运行的 SWF 文件和在放映文件中运行的 SWF 文件可以共享信息。（放映文件是以可作为独立应用程序运行的格式保存的 SWF 文件 — 即，放映文件嵌入在可执行文件中，因此不需要安装 Flash Player。）

可以使用 LocalConnection 对象在使用不同 ActionScript 版本的 SWF 之间进行通信：

- ActionScript 3.0 LocalConnection 对象可以与使用 ActionScript 1.0 或 2.0 创建的 LocalConnection 对象进行通信。
- ActionScript 1.0 或 2.0 LocalConnection 对象可以与使用 ActionScript 3.0 创建的 LocalConnection 对象进行通信。

Flash Player 可自动处理不同版本 LocalConnection 对象间的通信。

最简便的 LocalConnection 对象使用方法是只允许位于同一个域中的 LocalConnection 对象之间进行通信。这样，您就不必担心安全方面的问题。但如果您需要在不同域之间进行通信，则有多种方法来实现安全性措施。有关详细信息，请参阅《ActionScript 3.0 语言和组件参考》中有关 `send()` 方法的 `connectionName` 参数的讨论以及 LocalConnection 类列表中的 `allowDomain()` 和 `domain` 条目。

| 提示 | 可以使用 LocalConnection 对象在一个 SWF 文件中收发数据，但是 Adobe 不建议这样做。相反，您应使用共享对象。 |
| --- | --- |

可以使用三种方式将回调方法添加到 LocalConnection 对象中：

- 使 LocalConnection 类成为子类，并添加方法。
- 将 `LocalConnection.client` 属性设置为实现方法的对象。
- 创建扩展 LocalConnection 的动态类，并动态附加方法。

添加回调方法的第一种方式是扩展 LocalConnection 类。您在自定义类中定义方法，而不是将它们动态添加到 LocalConnection 实例中。下面的代码说明了此方式：

```
package
{
  import flash.net.LocalConnection;
  public class CustomLocalConnection extends LocalConnection
  {
    public function CustomLocalConnection(connectionName:String)
    {
      try
      {
        connect(connectionName);
      }
      catch (error:ArgumentError)
      {
        // 已创建 / 连接服务器
      }
    }
    public function onMethod(timeString:String):void
    {
      trace("onMethod called at: " + timeString);
    }
  }
}
```

要新建 DynamicLocalConnection 类的实例，可以使用下面的代码：

```
var serverLC:CustomLocalConnection;
serverLC = new CustomLocalConnection("serverName");
```

添加回调方法的第二种方式是使用 `LocalConnection.client` 属性。这包括创建自定义类和将新实例分配给 `client` 属性，如下面的代码所示：

```
var lc:LocalConnection = new LocalConnection();
lc.client = new CustomClient();
```

`LocalConnection.client` 属性指示应调用的对象回调方法。在上面的代码中，`client` 属性设置为自定义类 CustomClient 的新实例。`client` 属性的默认值是当前 LocalConnection 实例。如果有两个具有同一方法集但是作用不同的数据处理函数，则可以使用 `client` 属性 — 例如，在某个应用程序中，一个窗口中的按钮切换第二个窗口中的视图。

要创建 CustomClient 类，可以使用下面的代码：

```
package
{
  public class CustomClient extends Object
  {
    public function onMethod(timeString:String):void
    {
      trace("onMethod called at: " + timeString);
    }
  }
}
```

添加回调方法的第三种方式是创建动态类并动态附加该方法，这与在早期版本的 ActionScript 中使用 LocalConnection 类非常相似，如下面的代码所示：

```
import flash.net.LocalConnection;
dynamic class DynamicLocalConnection extends LocalConnection {}
```

通过使用下面的代码，可以将回调方法动态添加到此类中：

```
var connection:DynamicLocalConnection = new DynamicLocalConnection();
connection.onMethod = this.onMethod;
// 在此处添加代码。
public function onMethod(timeString:String):void
{
  trace("onMethod called at: " + timeString);
}
```

不建议使用上面这种添加回调方法的方式，因为该代码不是非常易于移植。另外，使用此方法创建本地连接会导致性能问题，因为访问动态属性比访问密封属性慢得多。

## 在两个 Flash Player 实例之间发送消息

可以使用 LocalConnection 类在 Flash Player 的不同实例之间进行通信。例如，可以在网页上有多个 Flash Player 实例，或者让 Flash Player 实例从弹出窗口中的 Flash Player 实例检索数据。

下面的代码定义一个本地连接对象，该对象作为服务器使用，接受来自其它 Flash Player 实例的传入调用：

```
package
{
  import flash.net.LocalConnection;
  import flash.display.Sprite;
  public class ServerLC extends Sprite
  {
    public function ServerLC()
    {
      var lc:LocalConnection = new LocalConnection();
      lc.client = new CustomClient1();
      try
      {
        lc.connect("conn1");
      }
      catch (error:Error)
      {
        trace("error:: already connected");
      }
    }
  }
}
```

此代码先创建一个名为 `lc` 的 LocalConnection 对象，然后将 `client` 属性设置为自定义类 CustomClient1。当另一个 Flash Player 实例调用此本地连接实例中的某方法时，Flash Player 在 CustomClient1 类中查找该方法。

当 Flash Player 实例连接到此 SWF 文件并尝试调用指定本地连接的任何方法时，系统会将请求发送到 `client` 属性指定的类（该属性被设置为 CustomClient1 类）：

```
package
{
  import flash.events.*;
  import flash.system.fscommand;
  import flash.utils.Timer;
  public class CustomClient1 extends Object
  {
    public function doMessage(value:String = ""):void
    {
      trace(value);
    }
    public function doQuit():void
```

```
    {
      trace("quitting in 5 seconds");
      this.close();
      var quitTimer:Timer = new Timer(5000, 1);
      quitTimer.addEventListener(TimerEvent.TIMER, closeHandler);
    }
    public function closeHandler(event:TimerEvent):void
    {
      fscommand("quit");
    }
  }
}
```

要创建 LocalConnection 服务器，请调用 `LocalConnection.connect()` 方法并提供唯一的连接名称。如果已存在具有指定名称的连接，则会生成 ArgumentError 错误，指出由于已经连接了该对象，连接尝试失败。

下面的片断说明如何使用名称 `conn1` 创建新的套接字连接：

```
try
{
  connection.connect("conn1");
}
catch (error:ArgumentError)
{
  trace("Error! Server already exists\n");
}
```

**注意** 在早期版本的 ActionScript 中，如果连接名称已被使用，`LocalConnection.connect()` 方法则会返回一个布尔值。在 ActionScript 3.0 中，如果该名称已被使用，则生成错误。

从辅助 SWF 文件连接到主 SWF 文件需要在发送方 LocalConnection 对象中新建 LocalConnection 对象，然后使用连接名称和要执行的方法名称来调用 `LocalConnection.send()` 方法。例如，要连接到早先创建的 LocalConnection 对象，可以使用下面的代码：

```
sendingConnection.send("conn1", "doQuit");
```

此代码使用连接名称 `conn1` 连接到现有 LocalConnection 对象，并调用远程 SWF 文件中的 `doQuit()` 方法。如果想要将参数发送到远程 SWF 文件，可以在 `send()` 方法中的方法名称后指定附加参数，如下面的片断所示：

```
sendingConnection.send("conn1", "doMessage", "Hello world");
```

# 连接到不同域中的 SWF 文档

要只允许从特定域进行通信，可以调用 LocalConnection 类的 allowDomain() 或 allowInsecureDomain() 方法，并传递包含允许访问此 LocalConnection 对象的一个或多个域的列表。

在早期版本的 ActionScript 中，LocalConnection.allowDomain() 和 LocalConnection.allowInsecureDomain() 是必须由开发人员实现的、且必须返回布尔值的回调方法。在 ActionScript 3.0 中，LocalConnection.allowDomain() 和 LocalConnection.allowInsecureDomain() 都是内置方法，开发人员可以像调用 Security.allowDomain() 和 Security.allowInsecureDomain() 那样调用这两个内置方法，传递要允许的一个或多个域的名称。

可以向 LocalConnection.allowDomain() 和 LocalConnection.allowInsecureDomain() 方法传递两个特殊值：* 和 localhost。星号值 (*) 表示允许从所有域访问。字符串 localhost 允许从本地安装的 SWF 文件调用 SWF 文件。

Flash Player 8 对本地 SWF 文件引入了安全限制。可以访问 Internet 的 SWF 文件还不能访问本地文件系统。如果指定 localhost，则任何本地 SWF 文件都可以访问 SWF 文件。如果 LocalConnection.send() 方法试图从调用代码没有访问权限的安全沙箱与 SWF 文件进行通信，则会调度 securityError 事件 (SecurityErrorEvent.SECURITY_ERROR)。要解决此错误，可以在接收方的 LocalConnection.allowDomain() 方法中指定调用方的域。

如果仅在同一个域中的 SWF 文件之间实现通信，可以指定一个不以下划线 (_) 开头且不指定域名的 connectionName 参数 （例如 myDomain:connectionName）。在 LocalConnection.connect(connectionName) 命令中使用相同的字符串。

如果要实现不同域中的 SWF 文件之间的通信，可以指定一个以下划线开头的 connectionName 参数。指定下划线使具有接收方 LocalConnection 对象的 SWF 文件更易于在域之间移植。下面是两种可能的情形：

- 如果 connectionName 字符串不以下划线开头，则 Flash Player 会添加一个包含超级域名称和一个冒号的前缀 （例如 myDomain:connectionName） 。虽然这可以确保您的连接不会与其它域中具有同一名称的连接冲突，但任何发送方 LocalConnection 对象都必须指定此超级域 （例如 myDomain:connectionName）。如果将具有接收方 LocalConnection 对象的 SWF 文件移动到另一个域中，Flash Player 会更改前缀，以反映新的超级域（例如 anotherDomain:connectionName）。必须手动编辑所有发送方 LocalConnection 对象，以指向新超级域。
- 如果 connectionName 字符串以下划线开头（例如 _connectionName），Flash Player 不会向该字符串添加前缀。这意味着接收方和发送方 LocalConnection 对象都将使用相同的 connectionName 字符串。如果接收方对象使用 LocalConnection.allowDomain() 来指定可以接受来自任何域的连接，则可以将具有接收方 LocalConnection 对象的 SWF 文件移动到另一个域，而无需更改任何发送方 LocalConnection 对象。

# 套接字连接

在 ActionScript 3.0 中，可以使用两种不同类型的套接字连接：XML 套接字连接和二进制套接字连接。使用 XML 套接字，可以连接到远程服务器并创建服务器连接，该连接在显式关闭之前一直保持打开。这样，无需不断打开新服务器连接，就可以在服务器与客户端之间交换 XML 数据。使用 XML 套接字服务器的另一个好处是用户不需要显式请求数据。您无需请求即可从服务器发送数据，并且可以将数据发送到连接到 XML 套接字服务器的每个客户端。

二进制套接字连接与 XML 套接字类似，不同之处是客户端与服务器不需要专门交换 XML 数据包，连接可以将数据作为二进制信息传输。这样，您就可以连接到各种各样的服务，包括邮件服务器 （POP3、 SMTP 和 IMAP）和新闻服务器 (NNTP)。

## Socket 类

ActionScript 3.0 中引入的 Socket 类使 ActionScript 可以建立套接字连接并读取和写入原始二进制数据。它与 XMLSocket 类相似，但没有指定接收和传输数据的格式。使用二进制协议的服务器互操作时，Socket 类与非常有用。使用二进制套接字连接，可以编写允许用一些不同的 Internet 协议（例如 POP3、 SMTP、 IMAP 和 NNTP）进行交互的代码。反过来，这又会使 Flash Player 能够连接到邮件和新闻服务器。

Flash Player 可通过使用服务器的二进制协议直接与该服务器连接。某些服务器使用 big-endian 字节顺序，某些服务器则使用 little-endian 字节顺序。 Internet 上的大多数服务器使用 big-endian 字节顺序，因为“网络字节顺序”为 big-endian。little-endian 字节顺序很常用，因为 Intel x86® 体系结构使用该字节顺序。您应使用与收发数据的服务器的字节顺序相匹配的 endian 字节顺序。默认情况下，IDataInput 和 IDataOutput 接口执行的所有操作和实现这些接口的类（ByteArray、 Socket 和 URLStream）都以 big-endian 格式编码；即，最高有效字节位于前面。这样做是为了匹配 Java 和官方网络字节顺序。要更改是使用 big-endian 还是使用 little-endian 字节顺序，可以将 `endian` 属性设置为 `Endian.BIG_ENDIAN` 或 `Endian.LITTLE_ENDIAN`。

| 提示 | Socket 类继承 IDataInput 和 IDataOutput 接口 （位于 flash.utils 包中）实现的所有方法，应使用这些方法从 Socket 读取数据和向其中写入数据。 |
|---|---|

# XMLSocket 类

ActionScript 提供了一个内置的 XMLSocket 类，使用它可以打开与服务器的持续连接。这种打开的连接消除了反应时间问题，它通常用于实时的应用程序，例如聊天应用程序或多人游戏。传统的基于 HTTP 的聊天解决方案频繁轮询服务器，并使用 HTTP 请求来下载新的消息。与此相对照，XMLSocket 聊天解决方案保持与服务器的开放连接，这一连接允许服务器即时发送传入的消息，而无需客户端发出请求。

要创建套接字连接，必须创建服务器端应用程序来等待套接字连接请求，然后向 SWF 文件发送响应。可以使用 Java、Python 或 Perl 程语言来编写这种类型的服务器端应用程序。要使用 XMLSocket 类，服务器计算机必须运行可识别 XMLSocket 类使用的协议的守护程序。下面的列表说明了该协议：

- 通过全双工 TCP/IP 流套接字连接发送 XML 消息。
- 每个 XML 消息都是一个完整的 XML 文档，以一个零 (0) 字节结束。
- 通过一个 XMLSocket 连接发送和接收的 XML 消息的数量没有限制。



XMLSocket 类不能自动穿过防火墙，因为 XMLSocket 没有 HTTP 隧道功能 （这与实时消息传递协议 (RTMP) 不同）。如果您需要使用 HTTP 隧道，应考虑改用 Flash Remoting 或 Flash Media Server （支持 RTMP）。

XMLSocket 对象连接到服务器的方式和位置受以下限制：

- `XMLSocket.connect()` 方法只能连接到端口号大于或等于 1024 的 TCP 端口。这种限制的一个后果是，与 `XMLSocket` 对象进行通信的服务器守护程序也必须分配到端口号大于或等于 1024 的端口。端口号小于 1024 的端口通常用于系统服务 （如 FTP (21)、Telnet (23)、SMTP (25)、HTTP (80) 和 POP3 (110)），因此，出于安全方面的考虑，禁止 XMLSocket 对象使用这些端口。这种端口号方面的限制可以减少不恰当地访问和滥用这些资源的可能性。
- `XMLSocket.connect()` 方法只能连接到 SWF 文件所在域中的计算机。这一限制不适用于在本地磁盘外运行的 SWF 文件。（此限制与 `URLLoader.load()` 的安全规则相同。）要连接到在 SWF 所在域之外的其它域中运行的服务器守护程序，可以在该服务器上创建一个允许从特定域进行访问的安全策略文件。



将服务器设置为与 XMLSocket 对象进行通信可能会遇到一些困难。如果您的应用程序不需要进行实时交互，请使用 URLLoader 类，而不要使用 XMLSocket 类。

可以使用 XMLSocket 类的 `XMLSocket.connect()` 和 `XMLSocket.send()` 方法，通过套接字连接与服务器之间传输 XML。 `XMLSocket.connect()` 方法与 Web 服务器端口建立套接字连接。 `XMLSocket.send()` 方法将 XML 对象传递给套接字连接中指定的服务器。

调用 `XMLSocket.connect()` 方法时， Flash Player 打开与服务器的 TCP/IP 连接，并使该连接保持打开状态，直到发生以下任一事件：

- XMLSocket 类的 `XMLSocket.close()` 方法被调用。
- 对 XMLSocket 对象的引用不再存在。
- Flash Player 退出。
- 连接中断 （例如，调制解调器断开连接）。

## 创建并连接到 Java XML 套接字服务器

下面的代码说明一个用 Java 编写的简单 XMLSocket 服务器，该服务器接受传入连接并在命令提示窗口中显示收到的消息。虽然从命令行启动服务器时可以指定其它端口号，但默认情况下，在本地计算机上的 8080 端口创建新服务器。

新建一个文本文档并添加下面的代码：

```
import java.io.*;
import java.net.*;

class SimpleServer
{
  private static SimpleServer server;
  ServerSocket socket;
  Socket incoming;
  BufferedReader readerIn;
  PrintStream printOut;

  public static void main(String[] args)
  {
    int port = 8080;

    try
    {
      port = Integer.parseInt(args[0]);
    }
    catch (ArrayIndexOutOfBoundsException e)
    {
      // 捕获异常并继续操作。
    }

    server = new SimpleServer(port);
  }

  private SimpleServer(int port)
  {
    System.out.println(">> Starting SimpleServer");
    try
    {
```

```
        socket = new ServerSocket(port);
        incoming = socket.accept();
        readerIn = new BufferedReader(new
    InputStreamReader(incoming.getInputStream()));
        printOut = new PrintStream(incoming.getOutputStream());
        printOut.println("Enter EXIT to exit.\r");
        out("Enter EXIT to exit.\r");
        boolean done = false;
        while (!done)
        {
          String str = readerIn.readLine();
          if (str == null)
          {
            done = true;
          }
          else
          {
            out("Echo: " + str + "\r");
            if(str.trim().equals("EXIT"))
            {
              done = true;
            }
          }
          incoming.close();
        }
      }
      catch (Exception e)
      {
        System.out.println(e);
      }
    }

    private void out(String str)
    {
      printOut.println(str);
      System.out.println(str);
    }
}
```

将文档保存到硬盘，命名为 SimpleServer.java 并使用 Java 编译器对其进行编译，这会创建一个名为 SimpleServer.class 的 Java 类文件。

您可以通过打开命令提示并键入 `java SimpleServer` 来启动 XMLSocket 服务器。SimpleServer.class 文件可以位于本地计算机或网络上的任何位置，不需要放置在 Web 服务器的根目录中。

> **提示** 如果由于文件没有位于 Java classpath 中而无法启动服务器，请尝试使用 `java -Classpath . SimpleServer` 启动服务器。

要从 ActionScript 应用程序连接到 XMLSocket，需要新建一个 XMLSocket 类实例，并在传递主机名和端口号时调用 `XMLSocket.connect()` 方法，如下所示：

```
var xmlsock:XMLSocket = new XMLSocket();
xmlsock.connect("127.0.0.1", 8080);
```

如果 `XMLSocket.connect()` 调用尝试连接到调用方安全沙箱以外的服务器，或者连接到低于 1024 的端口，则会发生 `securityError` (flash.events.SecurityErrorEvent) 事件。

只要从服务器接收数据，就会调度 `data` 事件 (`flash.events.DataEvent.DATA`)：

```
xmlsock.addEventListener(DataEvent.DATA, onData);
private function onData(event:DataEvent):void
{
  trace("[" + event.type + "] " + event.data);
}
```

要将数据发送到 XMLSocket 服务器，可以使用 `XMLSocket.send()` 方法并传递 XML 对象或字符串。Flash Player 将提供的参数转换为 String 对象，并将内容发送到 XMLSocket 服务器（后跟零 (0) 字节）：

```
xmlsock.send(xmlFormattedData);
```

`XMLSocket.send()` 方法不返回指示是否成功传输数据的值。如果尝试发送数据时发生错误，将引发 IOError 错误。

| 提示 | 发送到 XML 套接字服务器的每条消息必须以换行符 (`\n`) 结束。 |
|---|---|

# 存储本地数据

共享对象（有时称为“Flash cookie”）是一个数据文件，您访问的站点可能会在您的计算机上创建该文件。共享对象通常用于增强您的 Web 浏览体验 — 例如，使用它可以个性化经常访问的网站的外观。共享对象本身不能对您计算机上的数据进行任何操作，也不能使用这些数据进行任何操作。更重要的是，共享对象永远不能访问或记住您的电子邮件地址或其它个人信息 — 除非您愿意提供这样的信息。

可以使用静态的 `SharedObject.getLocal()` 或 `SharedObject.getRemote()` 方法来创建新的共享对象实例。`getLocal()` 方法尝试加载仅对当前客户端可用的本地永久共享对象，而 `getRemote()` 方法则尝试加载可借助服务器（例如 Flash Media Server）跨多个客户端共享的远程共享对象。如果本地或远程共享对象不存在，则 `getLocal()` 和 `getRemote()` 方法将创建一个新的 SharedObject 实例。

下面的代码尝试加载名为 `test` 的本地共享对象。如果此共享对象不存在，将新建一个具有此名称的共享对象。

```
var so:SharedObject = SharedObject.getLocal("test");
trace("SharedObject is " + so.size + " bytes");
```

如果找不到名为 test 的共享对象，则新建一个 0 字节大小的共享对象。如果已经存在该共享对象，则返回其当前大小（以字节为单位）。

您可以通过向数据对象分配值来在共享对象中存储数据，如以下示例所示：

```
var so:SharedObject = SharedObject.getLocal("test");
so.data.now = new Date().time;
trace(so.data.now);
trace("SharedObject is " + so.size + " bytes");
```

如果已经存在名为 test 的共享对象和参数 now，则会覆盖现有值。您可以使用 SharedObject.size 属性来确定共享对象是否已存在，如以下示例所示：

```
var so:SharedObject = SharedObject.getLocal("test");
if (so.size == 0)
{
  // 共享对象不存在。
  trace("created...");
  so.data.now = new Date().time;
}
trace(so.data.now);
trace("SharedObject is " + so.size + " bytes");
```

上面的代码使用 size 参数确定具有指定名称的共享对象实例是否已存在。如果测试下面的代码，您会注意到每次运行代码时，都会重新创建共享对象。为了将共享对象保存到用户的硬盘驱动器，您必须显式地调用 SharedObject.flush() 方法，如以下示例所示：

```
var so:SharedObject = SharedObject.getLocal("test");
if (so.size == 0)
{
  // 共享对象不存在。
  trace("created...");
  so.data.now = new Date().time;
}
trace(so.data.now);
trace("SharedObject is " + so.size + " bytes");
so.flush();
```

使用 flush() 方法将共享对象写入用户的硬盘时，应仔细检查用户是否已使用 Flash Player 设置管理器 (www.macromedia.com/support/documentation/en/flashplayer/help/settings_manager07.html) 显式禁用了本地存储，如下例所示：

```
var so:SharedObject = SharedObject.getLocal("test");
trace("Current SharedObject size is " + so.size + " bytes.");
so.flush();
```

可以在共享对象的 data 属性中指定属性名称以从共享对象检索值。例如，如果运行下面的代码，Flash Player 将显示 SharedObject 实例是在多少分钟前创建的：

```
var so:SharedObject = SharedObject.getLocal("test");
if (so.size == 0)
{
  // 共享对象不存在。
  trace("created...");
  so.data.now = new Date().time;
}
var ageMS:Number = new Date().time - so.data.now;
trace("SharedObject was created " + Number(ageMS / 1000 / 60).toPrecision(2)
  + " minutes ago");
trace("SharedObject is " + so.size + " bytes");
so.flush();
```

第一次运行上面的代码时，将创建一个名为 test 的新 SharedObject 实例，其初始大小为 0 字节。由于初始大小为 0 字节，if 语句的计算结果为 true，且一个名为 now 的新属性会添加到共享对象中。可以用当前时间减去 now 属性的值来计算共享对象的存在时间。以后每次运行上面的代码时，共享对象的大小都应大于 0，且该代码将跟踪共享对象是在多少分钟前创建的。

## 显示共享对象的内容

值存储在共享对象中的 data 属性中。可以使用 for..in 循环来循环访问共享对象实例中的每个值，如以下示例所示：

```
var so:SharedObject = SharedObject.getLocal("test");
so.data.hello = "world";
so.data.foo = "bar";
so.data.timezone = new Date().timezoneOffset;
for (var i:String in so.data)
{
  trace(i + ":\t" + so.data[i]);
}
```

## 创建安全 SharedObject

当使用 getLocal() 或 getRemote() 创建本地或远程 SharedObject 时，有一个名为 secure 的可选参数，该参数确定对此共享对象的访问是否限于通过 HTTPS 连接传递的 SWF 文件。如果此参数设置为 true 且 SWF 文件是通过 HTTPS 传递的，Flash Player 将新建一个安全共享对象，或者获取对现有安全共享对象的引用。此安全共享对象只能由通过 HTTPS 传递的 SWF 文件来读取或写入，SWF 文件将调用 SharedObject.getLocal() 并将 secure 参数设置为 true。如果此参数设置为 false 且 SWF 文件是通过 HTTPS 传递的，Flash Player 将新建一个共享对象，或者获取对现有共享对象的引用。

还可由通过非 HTTPS 连接传递的 SWF 文件对此共享对象进行读取或写入。如果 SWF 文件是通过非 HTTPS 连接传递的，并且您尝试将此参数设置为 `true`，将无法创建新的共享对象 （或访问以前创建的安全共享对象），并会引发错误，并且共享对象设置为 `null`。如果尝试通过非 HTTPS 连接运行以下代码片段， `SharedObject.getLocal()` 方法将引发错误：

```
try
{
  var so:SharedObject = SharedObject.getLocal("contactManager", null, true);
}
catch (error:Error)
{
  trace("Unable to create SharedObject.");
}
```

无论此参数为何值，创建的共享对象的数量都接近域所允许的磁盘空间的总量。

# 处理文件上载和下载

可通过使用 FileReference 类，在客户端和服务器之间添加上载和下载文件的功能。将通过一个对话框（例如， Windows 操作系统中的 “打开”对话框）提示用户选择要上载的文件或用于下载的位置。

使用 ActionScript 创建的每个 FileReference 对象都引用用户硬盘上的一个文件。该对象的属性包含有关文件大小、类型、名称、创建日期和修改日期的信息。



仅 Mac OS 支持 `creator` 属性。所有其它平台都会返回 `null`。

可以通过两种方式创建 FileReference 类的实例。可以使用 `new` 运算符，如以下代码所示：

```
import flash.net.FileReference;
var myFileReference:FileReference = new FileReference();
```

或者，可以调用 `FileReferenceList.browse()` 方法，该方法在用户的系统中打开一个对话框，提示用户选择一个或多个要上载的文件，如果用户成功选择了一个或多个文件，将创建一个由 FileReference 对象组成的数组。每个 FileReference 对象表示用户在对话框中选择的一个文件。 FileReference 对象在 FileReference 属性 （例如 `name`、 `size` 或 `modificationDate`）中不包含任何数据，直到发生下列任一情况：

- 已调用 `FileReference.browse()` 方法或 `FileReferenceList.browse()` 方法，并且用户已经从文件选取器中选择了文件。
- 已调用 `FileReference.download()` 方法，并且用户已经从文件选取器中选择了文件。

注意

当执行下载时，在下载完成前只填充 `FileReference.name` 属性。下载完文件后，所有属性都将可用。

在执行对 `FileReference.browse()`、 `FileReferenceList.browse()` 或 `FileReference.download()` 方法的调用时，大多数播放器将继续 SWF 文件播放。

# FileReference 类

使用 FileReference 类可以在用户的计算机和服务器之间上载和下载文件。操作系统对话框会提示用户选择要上载的文件或用于下载的位置。每个 FileReference 对象都引用用户磁盘上的一个文件并且具有一些属性，这些属性包含有关文件大小、类型、名称、创建日期、修改日期以及创建者的信息。

FileReference 实例的创建方法有两种：

■ 使用 `new` 运算符和 FileReference 构造函数，如下所示：

```
var myFileReference:FileReference = new FileReference();
```

■ 调用 `FileReferenceList.browse()`，从而创建 FileReference 对象数组。

对于上载和下载操作， SWF 文件只能访问自己的域 （包括由跨域策略文件指定的任何域）内的文件。如果启动上载或下载的 SWF 与文件服务器不在同一个域中，则需要将策略文件放到文件服务器上。


一次只能执行一个 `browse()` 或 `download()` 操作，因为在任何时刻只能打开一个对话框。

处理文件上载的服务器脚本应收到包含下列元素的 HTTP `POST` 请求：

■ `Content-Type`，其值为 `multipart/form-data`。

■ `Content-Disposition`，其 `name` 属性设置为 "`Filedata`"，`filename` 属性设置为原始文件的名称。您可以通过在 `FileReference.upload()` 方法中传递 `uploadDataFieldName` 参数的值来指定自定义 `name` 属性。

■ 文件的二进制内容。

下面是一个 HTTP `POST` 请求范例：

```
POST /handler.asp HTTP/1.1
Accept: text/*
Content-Type: multipart/form-data;
boundary=----------Ij5ae0ae0KM7GI3KM7ei4cH2ei4gL6
User-Agent: Shockwave Flash
Host: www.mydomain.com
Content-Length: 421
Connection: Keep-Alive
Cache-Control: no-cache

------------Ij5ae0ae0KM7GI3KM7ei4cH2ei4gL6
Content-Disposition: form-data; name="Filename"

sushi.jpg
------------Ij5ae0ae0KM7GI3KM7ei4cH2ei4gL6
Content-Disposition: form-data; name="Filedata"; filename="sushi.jpg"
Content-Type: application/octet-stream

Test File
```

```
------------Ij5ae0ae0KM7GI3KM7ei4cH2ei4gL6
Content-Disposition: form-data; name="Upload"

Submit Query
------------Ij5ae0ae0KM7GI3KM7ei4cH2ei4gL6
(actual file data,,,)
```

下面的 HTTP POST 请求范例发送三个 POST 变量：api_sig、api_key 和 auth_token，并使用自定义上载数据字段名的值 “photo”：

```
POST /handler.asp HTTP/1.1
Accept: text/*
Content-Type: multipart/form-data;
boundary=----------Ij5ae0ae0KM7GI3KM7ei4cH2ei4gL6
User-Agent: Shockwave Flash
Host: www.mydomain.com
Content-Length: 421
Connection: Keep-Alive
Cache-Control: no-cache

------------Ij5GI3GI3ei4GI3ei4KM7GI3KM7KM7
Content-Disposition: form-data; name="Filename"

sushi.jpg
------------Ij5GI3GI3ei4GI3ei4KM7GI3KM7KM7
Content-Disposition: form-data; name="api_sig"

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
------------Ij5GI3GI3ei4GI3ei4KM7GI3KM7KM7
Content-Disposition: form-data; name="api_key"

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
------------Ij5GI3GI3ei4GI3ei4KM7GI3KM7KM7
Content-Disposition: form-data; name="auth_token"

XXXXXXXXXXXXXXXXXXXXXXX
------------Ij5GI3GI3ei4GI3ei4KM7GI3KM7KM7
Content-Disposition: form-data; name="photo"; filename="sushi.jpg"
Content-Type: application/octet-stream

(actual file data,,,)
------------Ij5GI3GI3ei4GI3ei4KM7GI3KM7KM7
Content-Disposition: form-data; name="Upload"

Submit Query
------------Ij5GI3GI3ei4GI3ei4KM7GI3KM7KM7--
```

## 将文件上载到服务器

要将文件上载到服务器，需要首先调用 browse() 方法，以允许用户选择一个或多个文件。接下来，当调用 FileReference.upload() 方法时，所选的文件将传输到服务器。如果用户使用 FileReferenceList.browse() 方法选择了多个文件，Flash Player 将创建一个称为 FileReferenceList.fileList 的所选文件数组。可随后使用 FileReference.upload() 方法分别上载每个文件。



使用 FileReference.browse() 方法时，您只能上载单个文件。要允许用户上载多个文件，您必须使用 FileReferenceList.browse() 方法。

虽然开发人员可以通过使用 FileFilter 类并将文件过滤器实例数组传递给 browse() 方法来指定一个或多个自定义文件类型过滤器，但是默认情况下，系统文件选取器对话框允许用户从本地计算机选取任何文件类型：

```
var imageTypes:FileFilter = new FileFilter("Images (*.jpg, *.jpeg, *.gif,
  *.png)", "*.jpg; *.jpeg; *.gif; *.png");
var textTypes:FileFilter = new FileFilter("Text Files (*.txt, *.rtf)",
  "*.txt; *.rtf");
var allTypes:Array = new Array(imageTypes, textTypes);
var fileRef:FileReference = new FileReference();
fileRef.browse(allTypes);
```

用户在系统文件选取器中选择文件并单击"打开"按钮后，将调度 Event.SELECT 事件。如果使用 FileReference.browse() 方法选择要上载的文件，则需要用下面的代码将该文件发送到 Web 服务器：

```
var fileRef:FileReference = new FileReference();
fileRef.addEventListener(Event.SELECT, selectHandler);
fileRef.addEventListener(Event.COMPLETE, completeHandler);
try
{
  var success:Boolean = fileRef.browse();
}
catch (error:Error)
{
  trace("Unable to browse for files.");
}
function selectHandler(event:Event):void
{
  var request:URLRequest = new URLRequest("http://www.[yourdomain].com/
  fileUploadScript.cfm")
  try
  {
    fileRef.upload(request);
  }
  catch (error:Error)
  {
    trace("Unable to upload file.");
```

```
    }
}
function completeHandler(event:Event):void
{
    trace("uploaded");
}
```

| 提示 | 您可以使用 URLRequest.method 和 URLRequest.data 属性，通过 FileReference.upload() 方法将数据发送到服务器，以使用 POST 或 GET 方法发送变量。 |
|---|---|

当您尝试使用 FileReference.upload() 方法上载文件时，可能调度下列任何事件：

- Event.OPEN：上载操作开始时调度。
- ProgressEvent.PROGRESS：文件上载操作期间定期调度。
- Event.COMPLETE：文件上载操作成功完成时调度。
- SecurityErrorEvent.SECURITY_ERROR：由于安全侵犯导致上载失败时调度。
- HTTPStatusEvent.HTTP_STATUS：由于 HTTP 错误导致上载失败时调度。
- IOErrorEvent.IO_ERROR：由于下列任何原因导致上载失败时调度：
    - 当 Flash Player 正在读取、写入或传输文件时发生输入 / 输出错误。
    - SWF 尝试将文件上载到要求身份验证（如用户名和密码）的服务器。在上载期间，Flash Player 不为用户提供输入密码的方法。
    - url 参数包含无效协议。FileReference.upload() 方法必须使用 HTTP 或 HTTPS。

| 提示 | Flash Player 不对需要身份验证的服务器提供完全支持。只有使用浏览器插件或 Microsoft ActiveX® 控件在浏览器中运行的 SWF 文件才可以提供一个对话框，来提示用户输入用户名和密码以进行身份验证，并且用户只有在通过身份验证后才能下载。对于使用插件或 ActiveX 控件进行的上载操作，或者使用独立或外部播放器进行的上载 / 下载操作，文件传输会失败。 |
|---|---|

如果在 ColdFusion 中创建服务器脚本以接受来自 Flash Player 的文件上载，可以使用类似如下的代码：

```
<cffile action="upload" filefield="Filedata" destination="#ExpandPath('./
    ')#" nameconflict="OVERWRITE" />
```

此 ColdFusion 代码上载 Flash Player 发送的文件，并将其保存到 ColdFusion 模板所在的目录中以覆盖具有相同名称的任何文件。上面的代码显示接受文件上载所需的最低代码量；不应在生产环境中使用此脚本。理想情况下，应添加数据验证，以确保用户只上载接受的文件类型（例如图像），而不是上载可能危险的服务器端脚本。

下面的代码使用 PHP 说明文件上载，并且包含数据验证。该脚本将上载目录中的上载文件数限制为 10，确保文件小于 200 KB，并且只允许 JPEG、 GIF 或 PNG 文件上载和保存到文件系统。

```
<?php
$MAXIMUM_FILESIZE = 1024 * 200; // 200KB
$MAXIMUM_FILE_COUNT = 10; // keep maximum 10 files on server
echo exif_imagetype($_FILES['Filedata']);
if ($_FILES['Filedata']['size'] <= $MAXIMUM_FILESIZE)
{
  move_uploaded_file($_FILES['Filedata']['tmp_name'], "./temporary/
  ".$_FILES['Filedata']['name']);
  $type = exif_imagetype("./temporary/".$_FILES['Filedata']['name']);
  if ($type == 1 || $type == 2 || $type == 3)
  {
    rename("./temporary/".$_FILES['Filedata']['name'], "./images/
  ".$_FILES['Filedata']['name']);
  }
  else
  {
    unlink("./temporary/".$_FILES['Filedata']['name']);
  }
}
$directory = opendir('./images/');
$files = array();
while ($file = readdir($directory))
{
  array_push($files, array('./images/'.$file, filectime('./images/
  '.$file)));
}
usort($files, sorter);
if (count($files) > $MAXIMUM_FILE_COUNT)
{
  $files_to_delete = array_splice($files, 0, count($files) -
  $MAXIMUM_FILE_COUNT);
  for ($i = 0; $i < count($files_to_delete); $i++)
  {
    unlink($files_to_delete[$i][0]);
  }
}
print_r($files);
closedir($directory);

function sorter($a, $b)
{
  if ($a[1] == $b[1])
  {
    return 0;
  }
  else
```

```
  {
    return ($a[1] < $b[1]) ? -1 : 1;
  }
}
?>
```

可以使用 POST 或 GET 请求方法将附加变量传递到上载脚本。要将附加 POST 变量发送到上载脚本，可以使用下面的代码：

```
var fileRef:FileReference = new FileReference();
fileRef.addEventListener(Event.SELECT, selectHandler);
fileRef.addEventListener(Event.COMPLETE, completeHandler);
fileRef.browse();
function selectHandler(event:Event):void
{
  var params:URLVariables = new URLVariables();
  params.date = new Date();
  params.ssid = "94103-1394-2345";
  var request:URLRequest = new URLRequest("http://www.yourdomain.com/
  FileReferenceUpload/fileupload.cfm");
  request.method = URLRequestMethod.POST;
  request.data = params;
  fileRef.upload(request, "Custom1");
}
function completeHandler(event:Event):void
{
  trace("uploaded");
}
```

上面的示例新建一个将传递到远程服务器端脚本的 URLVariables 对象。在早期版本的 ActionScript 中，可以通过在查询字符串中传递值来将变量传递到服务器上载脚本。在 ActionScript 3.0 中，可以使用 URLRequest 对象将变量传递到远程脚本，该对象允许您使用 POST 或 GET 方法传递数据；因此，可以更轻松和更清晰地传递较大数据集。为了指定是使用 GET 还是使用 POST 请求方法来传递变量，可以将 URLRequest.method 属性相应设置为 URLRequestMethod.GET 或 URLRequestMethod.POST。

在 ActionScript 3.0 中，还可以通过向 upload() 方法提供第二个参数来覆盖默认 Filedata 上载文件字段名称，如上面的示例所示 （该示例使用 Custom1 替换默认值 Filedata）。

默认情况下， Flash Player 不尝试发送测试上载，虽然您可以通过将值 true 作为第三个参数传递给 upload() 方法来覆盖此行为。测试上载的目的是检查实际文件上载是否会成功，如果需要服务器身份，还会检查服务器身份验证是否会成功。




目前，只在基于 Windows 的 Flash Player 上进行测试上载。

## 从服务器下载文件

您可以让用户使用 FileReference.download() 方法从服务器下载文件，该方法使用两个参数：request 和 defaultFileName。第一个参数是 URLRequest 对象，该对象包含要下载的文件的 URL。第二个参数是可选的，它允许您指定出现在下载文件对话框中的默认文件名。如果省略第二个参数 defaultFileName，则使用指定 URL 中的文件名。

下面的代码从 SWF 文档所在的目录下载名为 index.xml 的文件：

```
var request:URLRequest = new URLRequest("index.xml");
var fileRef:FileReference = new FileReference();
fileRef.download(request);
```

要将默认名称设置为 currentnews.xml 而非 index.xml，请指定 defaultFileName 参数，如以下片断所示：

```
var request:URLRequest = new URLRequest("index.xml");
var fileToDownload:FileReference = new FileReference();
fileToDownload.download(request, "currentnews.xml");
```

如果服务器文件名不直观，或者文件名是由服务器生成的，则文件重命名可能非常有用。另外，在使用服务器端脚本下载文件（而不是直接下载文件）时，最好显式指定 defaultFileName 参数。例如，如果有基于传递给它的 URL 变量下载特定文件的服务器端脚本，则需要指定 defaultFileName 参数。否则，下载文件的默认名称即是服务器端脚本的名称。

可以使用 download() 方法将数据发送至服务器，方法是将参数追加到要分析的服务器脚本的 URL。下面的 ActionScript 3.0 片断基于传递给 ColdFusion 脚本的参数下载文档：

```
package
{
  import flash.display.Sprite;
  import flash.net.FileReference;
  import flash.net.URLRequest;
  import flash.net.URLRequestMethod;
  import flash.net.URLVariables;

  public class DownloadFileExample extends Sprite
  {
    private var fileToDownload:FileReference;
    public function DownloadFileExample()
    {
      var request:URLRequest = new URLRequest();
      request.url = "http://www.[yourdomain].com/downloadfile.cfm";
      request.method = URLRequestMethod.GET;
      request.data = new URLVariables("id=2");
      fileToDownload = new FileReference();
      try
      {
        fileToDownload.download(request, "file2.txt");
      }
      catch (error:Error)
```

```
      {
        trace("Unable to download file.");
      }
    }
  }
}
```

下面的代码显示了 ColdFusion 脚本 download.cfm，该脚本根据 URL 变量的值从服务器下载两个文件之一：

```
<cfparam name="URL.id" default="1" />
<cfswitch expression="#URL.id#">
  <cfcase value="2">
    <cfcontent type="text/plain" file="#ExpandPath('two.txt')#"
  deletefile="No" />
  </cfcase>
  <cfdefaultcase>
    <cfcontent type="text/plain" file="#ExpandPath('one.txt')#"
  deletefile="No" />
  </cfdefaultcase>
</cfswitch>
```

## FileReferenceList 类

使用 FileReferenceList 类，用户可以选择一个或多个要上载到服务器端脚本的文件。文件上载是由 `FileReference.upload()` 方法处理的，必须对用户选择的每个文件调用此方法。

下面的代码创建两个 FileFilter 对象（`imageFilter` 和 `textFilter`），并在一个数组中将它们传递到 `FileReferenceList.browse()` 方法。这导致操作系统文件对话框显示两个可能的文件类型过滤器。

```
var imageFilter:FileFilter = new FileFilter("Image Files (*.jpg, *.jpeg,
  *.gif, *.png)", "*.jpg; *.jpeg; *.gif; *.png");
var textFilter:FileFilter = new FileFilter("Text Files (*.txt, *.rtf)",
  "*.txt; *.rtf");
var fileRefList:FileReferenceList = new FileReferenceList();
try
{
  var success:Boolean = fileRefList.browse(new Array(imageFilter,
  textFilter));
}
catch (error:Error)
{
  trace("Unable to browse for files.");
}
```

允许用户使用 FileReferenceList 类选择并上载一个或多个文件与使用 FileReference.browse() 选择文件是相同的，但 FileReferenceList 允许选择多个文件。上载多个文件时，要求您使用 FileReference.upload() 上载每个所选的文件，如以下代码所示：

```
var fileRefList:FileReferenceList = new FileReferenceList();
fileRefList.addEventListener(Event.SELECT, selectHandler);
fileRefList.browse();

function selectHandler(event:Event):void
{
  var request:URLRequest = new URLRequest("http://www.[yourdomain].com/
  fileUploadScript.cfm");
  var file:FileReference;
  var files:FileReferenceList = FileReferenceList(event.target);
  var selectedFileArray:Array = files.fileList;
  for (var i:uint = 0; i < selectedFileArray.length; i++)
  {
    file = FileReference(selectedFileArray[i]);
    file.addEventListener(Event.COMPLETE, completeHandler);
    try
    {
      file.upload(request);
    }
    catch (error:Error)
    {
      trace("Unable to upload files.");
    }
  }
}
function completeHandler(event:Event):void
{
  trace("uploaded");
}
```

由于将 Event.COMPLETE 事件添加到数组中每个单独的 FileReference 对象中，因此，Flash Player 在每个单独文件完成上载时将调用 completeHandler() 方法。

# 示例：构建 Telnet 客户端

Telnet 示例说明了使用 Socket 类连接远程服务器和传输数据的方法。该示例演示下列方法：

- 使用 Socket 类创建自定义 telnet 客户端
- 使用 ByteArray 对象将文本发送到远程服务器
- 处理从远程服务器收到的数据

要获取该范例的应用程序文件，请访问
www.adobe.com/go/learn_programmingAS3samples_flash_cn。可以在 Samples/Telnet 文件夹中找到 Telnet 应用程序文件。该应用程序包含以下文件：

| 文件 | 描述 |
|---|---|
| TelnetSocket.mxml | 包含 MXML 用户界面的主应用程序文件。 |
| com/example/programmingas3/Telnet/<br>Telnet.as | 为应用程序提供 Telnet 客户端功能，例如连接到远程服务器以及发送、接收和显示数据。 |

## Telnet 套接字应用程序概述

主 TelnetSocket.mxml 文件负责为整个应用程序创建用户界面 (UI)。

除了 UI，此文件还定义两个方法 `login()` 和 `sendCommand()`，以便将用户连接到指定的服务器。

下面的代码列出了主应用程序文件中的 ActionScript：

```
import com.example.programmingas3.socket.Telnet;

private var telnetClient:Telnet;
private function connect():void
{
  telnetClient = new Telnet(serverName.text, int(portNumber.text), output);
  console.title = "Connecting to " + serverName.text + ":" +
  portNumber.text;
  console.enabled = true;
}
private function sendCommand():void
{
  var ba:ByteArray = new ByteArray();
  ba.writeMultiByte(command.text + "\n", "UTF-8");
  telnetClient.writeBytesToSocket(ba);
  command.text = "";
}
```

第一行代码从自定义 com.example.programmingas.socket 包中导入 Telnet 类。第二行代码声明 Telnet 类的实例 telnetClient，稍后 connect() 方法将初始化该实例。接下来，声明 connect() 方法，该方法初始化先前声明的 telnetClient 变量。此方法传递用户指定的 telnet 服务器名称、telnet 服务器端口和对显示列表中 TextArea 组件的引用，其中，显示列表用于显示来自套接字服务器的文本响应。connect() 方法的最后两行设置 Panel 的 title 属性并启用 Panel 组件，该组件允许用户将数据发送到远程服务器。主应用程序文件中的最后一个方法 sendCommand() 用于将用户的命令作为 ByteArray 对象发送到远程服务器。

## Telnet 类概述

Telnet 类负责连接到远程 Telnet 服务器和发送 / 接收数据。

Telnet 类声明下列私有变量：

```
private var serverURL:String;
private var portNumber:int;
private var socket:Socket;
private var ta:TextArea;
private var state:int = 0;
```

第一个变量 serverURL 包含要连接到的用户指定的服务器地址。

第二个变量 portNumber 是 Telnet 服务器当前在其上运行的端口号。默认情况下，Telnet 服务在端口 23 上运行。

第三个变量 socket 是 Socket 实例，它将尝试连接到由 serverURL 和 portNumber 变量所定义的服务器。

第四个变量 ta 是对舞台上的 TextArea 组件实例的引用。此组件用于显示来自远程 Telnet 服务器的响应或者任何可能的错误消息。

最后一个变量 state 是数值，用于确定 Telnet 客户端支持哪些选项。

正如您之前所见， Telnet 类的构造函数由主应用程序文件中的 connect() 方法调用。

Telnet 构造函数采用三个参数：server、port 和 output。server 和 port 参数指定 Telnet 服务器在其上运行的服务器名称和端口号。最后一个参数 output 是对舞台上的 TextArea 组件实例的引用，将在舞台上向用户显示服务器输出。

```
public function Telnet(server:String, port:int, output:TextArea)
{
    serverURL = server;
    portNumber = port;
    ta = output;
    socket = new Socket();
    socket.addEventListener(Event.CONNECT, connectHandler);
    socket.addEventListener(Event.CLOSE, closeHandler);
    socket.addEventListener(ErrorEvent.ERROR, errorHandler);
    socket.addEventListener(IOErrorEvent.IO_ERROR, ioErrorHandler);
    socket.addEventListener(ProgressEvent.SOCKET_DATA, dataHandler);
```

```
  Security.loadPolicyFile("http://" + serverURL + "/crossdomain.xml");
  try
  {
    msg("Trying to connect to " + serverURL + ":" + portNumber + "\n");
    socket.connect(serverURL, portNumber);
  }
  catch (error:Error)
  {
    msg(error.message + "\n");
    socket.close();
  }
}
```

## 向套接字写入数据

要向套接字连接写入数据，可以调用 Socket 类中的任何写入方法（例如 writeBoolean()、writeByte()、writeBytes() 或 writeDouble()），然后使用 flush() 方法刷新输出缓冲区中的数据。在 Telnet 服务器中，使用 writeBytes() 方法向套接字连接中写入数据，该方法将字节数组作为参数，并将其发送到输出缓冲区。writeBytesToSocket() 方法如下：

```
public function writeBytesToSocket(ba:ByteArray):void
{
  socket.writeBytes(ba);
  socket.flush();
}
```

此方法是由主应用程序文件中的 sendCommand() 方法调用的。

## 显示来自套接字服务器的消息

当从套接字服务器接收消息，或者发生事件时，将调用自定义 msg() 方法。此方法将一个字符串追加到舞台上的 TextArea，并调用自定义 setScroll() 方法，该方法导致 TextArea 组件滚动到最底端。msg() 方法如下：

```
private function msg(value:String):void
{
  ta.text += value;
  setScroll();
}
```

如果您未将 TextArea 组件的内容设置为自动滚动，则用户需要手动拖动文本区域中的滚动条才能看到来自服务器的最新响应。

## 滚动 TextArea 组件

`setScroll()` 方法包含一行 ActionScript，用于垂直滚动 TextArea 组件内容，以便用户可以查看返回文本的最后一行。下面的片断显示了 `setScroll()` 方法：

```
public function setScroll():void
{
   ta.verticalScrollPosition = ta.maxVerticalScrollPosition;
}
```

此方法设置 `verticalScrollPosition` 属性 （该属性是当前显示的最上面一行字符的行号），并将其设置为 `maxVerticalScrollPosition` 属性的值。

# 示例：上载和下载文件

FileIO 示例说明了在 Flash Player 中执行文件下载和上载的方法。这些方法包括：

- 将文件下载到用户的计算机
- 将文件从用户的计算机上载到服务器
- 取消正在进行的下载
- 取消正在进行的上载

要获取该范例的应用程序文件，请访问
www.adobe.com/go/learn_programmingAS3samples_flash_cn。在 Samples/FileIO 文件夹中可以找到 FileIO 应用程序文件。该应用程序包含以下文件：

| 文件 | 描述 |
|---|---|
| FileIO.fla<br>或<br>FileIO.mxml | Flash 或 Flex 中的主应用程序文件 （分别为 FLA 和 MXML）。 |
| com/example/programmingas3/fileio/FileDownload.as | 一个类，包含用于从服务器下载文件的方法。 |
| com/example/programmingas3/fileio/FileUpload.as | 一个类，包含用于将文件上载到服务器的方法。 |

## FileIO 应用程序概述

FileIO 应用程序包含用户界面，使用户可以使用 Flash Player 上载或下载文件。该应用程序首先定义一对自定义组件 FileUpload 和 FileDownload，可以在 com.example.programmingas3.fileio 包中找到这两个组件。每个自定义组件调度其 `contentComplete` 事件后，将调用该组件的 `init()` 方法，此方法传递对 ProgressBar 和 Button 组件实例的引用，从而使用户可以看到文件的上载或下载进度或者取消正在进行的文件传输。

FileIO.mxml 文件中的相关代码如下所示（请注意，在 Flash 版本中， FLA 文件包含放在舞台上的组件，这些组件的名称与此步骤中介绍的 Flex 组件的名称相匹配）：

```
<example:FileUpload id="fileUpload"
  creationComplete="fileUpload.init(uploadProgress, cancelUpload);" />
<example:FileDownload id="fileDownload"
  creationComplete="fileDownload.init(downloadProgress, cancelDownload);" />
```

下面的代码显示 “上载文件”面板，其中包含一个进度条和两个按钮。第一个按钮 `startUpload` 调用 `FileUpload.startUpload()` 方法，该方法调用 `FileReference.browse()` 方法。下面的摘选显示了用于 “上载文件”面板的代码：

```
<mx:Panel title="Upload File" paddingTop="10" paddingBottom="10"
  paddingLeft="10" paddingRight="10">
  <mx:ProgressBar id="uploadProgress" label="" mode="manual" />
  <mx:ControlBar horizontalAlign="right">
    <mx:Button id="startUpload" label="Upload..."
  click="fileUpload.startUpload();" />
    <mx:Button id="cancelUpload" label="Cancel"
  click="fileUpload.cancelUpload();" enabled="false" />
  </mx:ControlBar>
</mx:Panel>
```

此代码将一个 ProgressBar 组件实例和两个 Button 组件按钮实例放在舞台上。当用户单击 “上载”按钮 (`startUpload`) 时，会启动一个操作系统对话框，允许用户选择要上载到远程服务器的文件。尽管用户开始文件上载时，另一个按钮 `cancelUpload` 会变为可用，并允许用户随时中断文件传输，但默认情况下禁用该按钮。

用于 “下载文件”面板的代码如下：

```
<mx:Panel title="Download File" paddingTop="10" paddingBottom="10"
  paddingLeft="10" paddingRight="10">
  <mx:ProgressBar id="downloadProgress" label="" mode="manual" />
  <mx:ControlBar horizontalAlign="right">
    <mx:Button id="startDownload" label="Download..."
  click="fileDownload.startDownload();" />
    <mx:Button id="cancelDownload" label="Cancel"
  click="fileDownload.cancelDownload();" enabled="false" />
  </mx:ControlBar>
</mx:Panel>
```

此代码与文件上载代码非常相似。当用户单击 “下载”按钮 (startDownload) 时，会调用 FileDownload.startDownload() 方法，从而开始下载 FileDownload.DOWNLOAD_URL 变量中指定的文件。在文件下载时，进度条进行更新，以显示文件下载百分比。用户可以随时通过单击 cancelDownload 按钮取消下载，这样会立即停止正在进行的文件下载。

## 从远程服务器下载文件

从远程服务器下载文件由 flash.net.FileReference 类和自定义 com.example.programmingas3.fileio.FileDownload 类处理。当用户单击 “下载”按钮时， Flash Player 开始下载 FileDownload 类的 DOWNLOAD_URL 变量中指定的文件。

FileDownload 类从定义 com.example.programmingas3.fileio 包中的 4 个变量开始，如以下代码所示：

```
/**
 * 对要下载到用户计算机的文件的 URL 进行硬编码。
 */
private const DOWNLOAD_URL:String = "http://www.yourdomain.com/
file_to_download.zip";

/**
 * 创建 FileReference 实例以处理文件下载。
 */
private var fr:FileReference;

/**
 * 定义对下载 ProgressBar 组件的引用。
 */
private var pb:ProgressBar;

/**
 * 定义对 “取消”按钮的引用，该按钮将立即停止
 * 当前正在进行的下载。
 */
private var btn:Button;
```

第一个变量 DOWNLOAD_URL 包含文件的路径，当用户单击主应用程序文件中的 “下载”按钮时，该文件会下载到用户的计算机上。

第二个变量 fr 是 FileReference 对象，该对象在 FileDownload.init() 方法中进行初始化，用于处理远程文件到用户计算机的下载。

最后两个变量 pb 和 btn 包含对舞台上的 ProgressBar 和 Button 组件实例的引用，它们由 FileDownload.init() 方法进行初始化。

## 初始化 FileDownload 组件

通过在 FileDownload 类中调用 init() 方法对 FileDownload 组件进行初始化。此方法使用两个参数 pb 和 btn，它们分别是 ProgressBar 和 Button 组件实例。

用于 init() 方法的代码如下所示：

```
/**
 * 设置对组件的引用，并添加 OPEN、
 * PROGRESS 和 COMPLETE 事件的侦听器。
 */
public function init(pb:ProgressBar, btn:Button):void
{
    // 设置对进度条和 “取消”按钮的引用，
    // 它们是从调用脚本传递的。
    this.pb = pb;
    this.btn = btn;

    fr = new FileReference();
    fr.addEventListener(Event.OPEN, openHandler);
    fr.addEventListener(ProgressEvent.PROGRESS, progressHandler);
    fr.addEventListener(Event.COMPLETE, completeHandler);
}
```

## 开始文件下载

当用户单击舞台上的 Download Button 组件实例时，startDownload() 方法启动文件下载进程。下面的摘选显示了 startDownload() 方法：

```
/**
 * 开始下载在 DOWNLOAD_URL 常量中指定的文件。
 */
public function startDownload():void
{
    var request:URLRequest = new URLRequest();
    request.url = DOWNLOAD_URL;
    fr.download(request);
}
```

首先，startDownload() 方法创建一个新的 URLRequest 对象，并将目标 URL 设置为 DOWNLOAD_URL 变量指定的值。接下来，调用 FileReference.download() 方法，将新创建的 URLRequest 对象作为参数传递。这会导致操作系统在用户计算机上显示一个对话框，提示用户选择一个位置以保存所请求的文档。用户选择了位置后，将调度 open 事件 (Event.OPEN) 并调用 openHandler() 方法。

openHandler() 方法设置 ProgressBar 组件的 label 属性的文本格式，并启用“取消”按钮，允许用户立即停止正在进行的下载。openHandler() 方法如下所示：

```
/**
 * 调度 OPEN 事件后，更改进度条的标签
 * 并启用“取消”按钮，以使用户能够中止
 * 下载操作。
 */
private function openHandler(event:Event):void
{
  pb.label = "DOWNLOADING %3%%";
  btn.enabled = true;
}
```

## 监视文件的下载进度

当从远程服务器向用户计算机下载文件时，会定期调度 progress 事件 (ProgressEvent.PROGRESS)。每当调度 progress 事件时，会调用 progressHandler() 方法并更新舞台上的 ProgressBar 组件实例。用于 progressHandler() 方法的代码如下所示：

```
/**
 * 在下载文件的同时，更新进度栏的状态。
 */
private function progressHandler(event:ProgressEvent):void
{
  pb.setProgress(event.bytesLoaded, event.bytesTotal);
}
```

进度事件包含两个属性（bytesLoaded 和 bytesTotal），它们用于更新舞台上的 ProgressBar 组件。这样，用户可以了解已完成下载的文件比例以及尚未下载的文件比例。用户可以通过单击进度条下的“取消”按钮随时终止文件传输。

如果文件下载成功，complete 事件 (Event.COMPLETE) 将调用 completeHandler() 方法，该方法通知用户文件已完成下载并禁用“取消”按钮。用于 completeHandler() 方法的代码如下所示：

```
/**
 * 在下载完成后，更改一次（也是最后一次）进度栏的标签
 * 并禁用“取消”按钮，因为下载
 * 已经完成。
 */
private function completeHandler(event:Event):void
{
  pb.label = "DOWNLOAD COMPLETE";
  btn.enabled = false;
}
```

## 取消文件下载

用户可以通过单击舞台上的 “取消” 按钮随时终止文件传输和停止下载任何更多字节。下面的摘选显示了用于取消下载的代码：

```
/**
 * 取消当前文件下载。
 */
public function cancelDownload():void
{
    fr.cancel();
    pb.label = "DOWNLOAD CANCELLED";
    btn.enabled = false;
}
```

首先，该代码立即停止文件传输以禁止下载其它任何数据。接下来，进度条的 label 属性进行更新，通知用户下载已成功取消。最后，禁用 “取消” 按钮，从而阻止用户再次尝试下载文件之前再次单击该按钮。

## 将文件上载到远程服务器上

文件上载进程与文件下载进程非常相似。FileUpload 类声明相同的四个变量，如以下代码所示：

```
private const UPLOAD_URL:String = "http://www.yourdomain.com/
    your_upload_script.cfm";
private var fr:FileReference;
private var pb:ProgressBar;
private var btn:Button;
```

与 `FileDownload.DOWNLOAD_URL` 变量不同，`UPLOAD_URL` 变量包含将从用户计算机上载文件的服务器端脚本的 URL。其余三个变量的作用与 FileDownload 类中的对应变量相同。

## 初始化 FileUpload 组件

FileUpload 组件包含 `init()` 方法，此方法从主应用程序调用。此方法使用两个参数 `pb` 和 `btn`，它们是对舞台上的 ProgressBar 和 Button 组件实例的引用。接下来，`init()` 方法初始化先前在 FileUpload 类中定义的 FileReference 对象。最后，此方法将四个事件侦听器分配给 FileReference 对象。用于 `init()` 方法的代码如下所示：

```
public function init(pb:ProgressBar, btn:Button):void
{
    this.pb = pb;
    this.btn = btn;

    fr = new FileReference();
    fr.addEventListener(Event.SELECT, selectHandler);
    fr.addEventListener(Event.OPEN, openHandler);
```

```
  fr.addEventListener(ProgressEvent.PROGRESS, progressHandler);
  fr.addEventListener(Event.COMPLETE, completeHandler);
}
```

## 开始文件上载

当用户单击舞台上的 “上载”按钮时，启动文件上载，此操作将调用 `FileUpload.startUpload()` 方法。此方法调用 FileReference 类的 `browse()` 方法，从而导致操作系统显示一个系统对话框，以提示用户选择要上载到远程服务器的文件。下面的摘选显示了用于 `startUpload()` 方法的代码：

```
public function startUpload():void
{
  fr.browse();
}
```

在用户选择要上载的文件后，将调度 `select` 事件 (`Event.SELECT`)，从而导致调用 `selectHandler()` 方法。`selectHandler()` 方法新建一个 URLRequest 对象，并将 `URLRequest.url` 属性设置为先前在代码中定义的 `UPLOAD_URL` 常量的值。最后，FileReference 对象将所选文件上载到指定的服务器端脚本。用于 `selectHandler()` 方法的代码如下所示：

```
private function selectHandler(event:Event):void
{
  var request:URLRequest = new URLRequest();
  request.url = UPLOAD_URL;
  fr.upload(request);
}
```

FileUpload 类中的其余代码与 FileDownload 类中定义的代码相同。如果用户想要在任何时间点终止上载，可以单击 “取消”按钮，该按钮将在进度条上设置标签并立即停止文件传输。每当调度 `progress` 事件 (`ProgressEvent.PROGRESS`) 时，都会更新进度条。同样，当完成上载时，进度条进行更新以通知用户文件已上载成功。然后，禁用 “取消”按钮，直到用户开始新的文件传输。

第 23 章

# 客户端系统环境

23

本章说明如何与用户的系统进行交互。它介绍了如何确定支持哪些功能，以及如何通过用户安装的输入法编辑器 (IME)（如果可用）生成多语言 SWF 文件。还将介绍应用程序域的典型用法。

## 目录



# 客户端系统环境基础知识

## 客户端系统环境简介

在构建更高级的 ActionScript 应用程序时，您可能会发现需要了解有关用户操作系统的详细信息 （和访问操作系统功能）。客户端系统环境是 flash.system 包中的类集合，可通过这些类来访问系统级功能，例如：

- 确定执行 SWF 时所在的应用程序和安全域
- 确定用户的 Flash Player 的功能，如屏幕大小（分辨率）；以及确定某项功能是否可用，如 mp3 音频
- 使用 IME 建立多语言站点
- 与 Flash Player 的容器 （可能是 HTML 页或容器应用程序）进行交互
- 将信息保存到用户的剪贴板中

flash.system 包还包括 IMEConversionMode 和 SecurityPanel 类。这两个类分别包含与 IME 和 Security 类一起使用的静态常数。

## 常见客户端系统环境任务

本章介绍了通过 ActionScript 使用客户端系统的以下常见任务：

- 确定应用程序使用的内存数量
- 将文本复制到用户的剪贴板中
- 确定用户计算机的功能，例如：
  - 屏幕分辨率、颜色、 DPI 以及像素高宽比
  - 操作系统
  - 支持音频流、视频流以及 mp3 回放
  - 安装的 Flash Player 是否为调试版
- 处理应用程序域：
  - 定义应用程序域
  - 将 SWF 文件的代码划分到应用程序域中
- 在应用程序中处理 IME：
  - 确定是否安装了 IME
  - 确定并设置 IME 转换模式
  - 对文本字段禁用 IME
  - 检测何时发生 IME 转换

## 重要概念和术语

以下参考列表包含将会在本章中使用的重要术语：

- 操作系统 (Operating system)：计算机上运行的主程序 （其它所有应用程序均运行在其中），如 Microsoft Windows、 Mac OS X 或 Linux®。
- 剪贴板 (Clipboard)：用于保存复制或剪切的文本或项目的操作系统容器，可从中将项目粘贴到应用程序中。
- 应用程序域 (Application domain)：用于将不同 SWF 文件中使用的类分开的机制，以便在 SWF 文件包含具有相同名称的不同类时，这些类不会彼此覆盖。
- IME（input method editor，输入法编辑器）：用于通过标准键盘输入复杂字符或符号的程序 （或操作系统工具）。
- 客户端系统：在编程术语中，“客户端”是指在单独计算机上运行并由单个用户使用的应用程序部分 （或整个应用程序）。“客户端系统”是指用户计算机上的基础操作系统。

## 完成本章中的示例

学习本章的过程中，您可能想要自己动手测试一些示例代码清单。本章中的所有代码清单均包括相应的 `trace()` 函数调用，用于写出所测试的值。要测试本章中的代码清单，请执行以下操作：

1. 创建一个空的 Flash 文档。
2. 在时间轴上选择一个关键帧。
3. 打开 “动作”面板，将代码清单复制到 “脚本”窗格中。
4. 使用 “控制” > “测试影片”运行程序。

   您将在 “输出”面板中看到该代码清单的 `trace` 函数的结果。

后面的某些代码清单更为复杂一些，并且是以类的形式编写的。要测试这些示例，请执行以下操作：

1. 创建一个空的 Flash 文档并将它保存到您的计算机上。
2. 创建一个新的 ActionScript 文件，并将它保存到 Flash 文档所在的目录中。文件名应与代码清单中的类的名称一致。例如，如果代码清单定义一个名为 SystemTest 的类，则使用名称 SystemTest.as 来保存 ActionScript 文件。
3. 将代码清单复制到 ActionScript 文件中并保存该文件。
4. 在 Flash 文档中，单击舞台或工作区的空白部分，以激活文档的 “属性”检查器。
5. 在“属性”检查器的“文档类”字段中，输入您从文本中复制的 ActionScript 类的名称。
6. 使用 “控制” > “测试影片”运行程序

   您将在 “输出”面板中看到该示例的结果。

测试示例代码清单的技术在第 53 页的 “测试本章内的示例代码清单”中有更详细的介绍。

# 使用 System 类

System 类包含的一些方法和属性可让您与用户的操作系统进行交互，并检索 Adobe Flash Player 的当前内存使用数据。 System 类的方法和属性还可用来侦听 `imeComposition` 事件，指示 Flash Player 使用用户的当前代码页加载外部文本文件或按 Unicode 进行加载，或者设置用户剪贴板的内容。

## 在运行时获取有关用户系统的数据

通过检查 `System.totalMemory` 属性，可以确定 Flash Player 当前所用的内存数量（以字节为单位）。该属性可让您监视内存使用情况，并根据内存级别的更改方式优化应用程序。例如，如果特定视觉效果导致内存使用量大幅增加，您可能需要考虑修改此效果或将其完全消除。

`System.ime` 属性是对当前安装的输入法编辑器 (IME) 的引用。该属性允许使用 `addEventListener()` 方法来侦听 `imeComposition` 事件 (`flash.events.IMEEvent.IME_COMPOSITION`)。

System 类中的第三个属性是 `useCodePage`。如果将 `useCodePage` 设置为 `true`， Flash Player 将使用运行播放器的操作系统的传统代码页来加载外部文本文件。如果将此属性设置为 `false`，则 Flash Player 按 Unicode 解释外部文件。

如果将 `System.useCodePage` 设置为 `true`，请记住，运行播放器的操作系统的传统代码页中必须包括在外部文本文件中使用的字符，这样才能显示文本。例如，如果您加载了一个包含中文字符的外部文本文件，则这些字符不能显示在使用英文 Windows 代码页的系统上，因为该代码页不包括中文字符。

要确保所有平台上的用户都能查看 SWF 文件中使用的外部文本文件，应将所有外部文本文件按 Unicode 进行编码，并将 `System.useCodePage` 设置保留为默认设置 `false`。这样，Flash Player 就会将文本解释为 Unicode。

## 将文本保存到剪贴板

System 类包含一个名为 `setClipboard()` 的方法，它允许 Flash Player 使用指定的字符串来设置用户剪贴板的内容。出于安全方面的考虑，不存在 `Security.getClipboard()` 方法，因为此类方法可能允许恶意站点访问最近复制到用户剪贴板中的数据。

以下代码说明出现安全错误时如何将错误消息复制到用户剪贴板。如果用户要报告应用程序的潜在错误，则错误消息会很有用。

```
private function securityErrorHandler(event:SecurityErrorEvent):void
{
  var errorString:String = "[" + event.type + "] " + event.text;
  trace(errorString);
  System.setClipboard(errorString);
}
```

# 使用 Capabilities 类

开发人员可通过 Capabilities 类来确定正在运行 SWF 文件的环境。使用 Capabilities 类的各种属性，可以查明用户系统的分辨率、用户的系统是否支持辅助功能软件、用户操作系统的语言以及当前安装的 Flash Player 的版本。

通过检查 Capabilities 类的属性，可以自定义应用程序，使其与特定用户环境更好地配合使用。例如，通过检查 `Capabilities.screenResolutionX` 和 `Capabilities.screenResolutionY` 属性，可以确定用户系统所使用的显示分辨率以及决定最合适的视频大小。或者，在尝试加载外部 mp3 文件之前，可以检查 `Capabilities.hasMP3` 属性以查看用户系统是否支持 mp3 回放。

以下代码使用正则表达式分析客户端所使用的 Flash Player 版本：

```
var versionString:String = Capabilities.version;
var pattern:RegExp = /^(\w*) (\d*),(\d*),(\d*),(\d*)$/;
var result:Object = pattern.exec(versionString);
if (result != null)
{
  trace("input: " + result.input);
  trace("platform: " + result[1]);
  trace("majorVersion: " + result[2]);
  trace("minorVersion: " + result[3]);
  trace("buildNumber: " + result[4]);
  trace("internalBuildNumber: " + result[5]);
}
else
{
  trace("Unable to match RegExp.");
}
```

如果要向某个服务器端脚本发送用户的系统功能，以便将信息存储在数据库中，则可以使用以下 ActionScript 代码：

```
var url:String = "log_visitor.cfm";
var request:URLRequest = new URLRequest(url);
request.method = URLRequestMethod.POST;
request.data = new URLVariables(Capabilities.serverString);
var loader:URLLoader = new URLLoader(request);
```

# 使用 ApplicationDomain 类

ApplicationDomain 类的用途是存储 ActionScript 3.0 定义表。SWF 文件中的所有代码被定义为存在于应用程序域中。可以使用应用程序域划分位于同一个安全域中的类。这允许同一个类存在多个定义，并且还允许子级重用父级定义。

在使用 Loader 类 API 加载用 ActionScript 3.0 编写的外部 SWF 文件时，可以使用应用程序域。（请注意，在加载图像或用 ActionScript 1.0 或 ActionScript 2.0 编写的 SWF 文件时不能使用应用程序域。）包含在已加载类中的所有 ActionScript 3.0 定义都存储在应用程序域中。加载 SWF 文件时，通过将 LoaderContext 对象的 `applicationDomain` 参数设置为 `ApplicationDomain.currentDomain`，可以指定文件包含在 Loader 对象所在的相同应用程序域中。通过将加载的 SWF 文件放在同一个应用程序域中，可以直接访问它的类。如果加载的 SWF 文件包含嵌入的媒体（可通过其关联的类名称访问），或者您要访问加载的 SWF 文件的方法，则这种方式会很有用。如以下示例所示：

```
package
{
  import flash.display.Loader;
  import flash.display.Sprite;
  import flash.events.*;
```

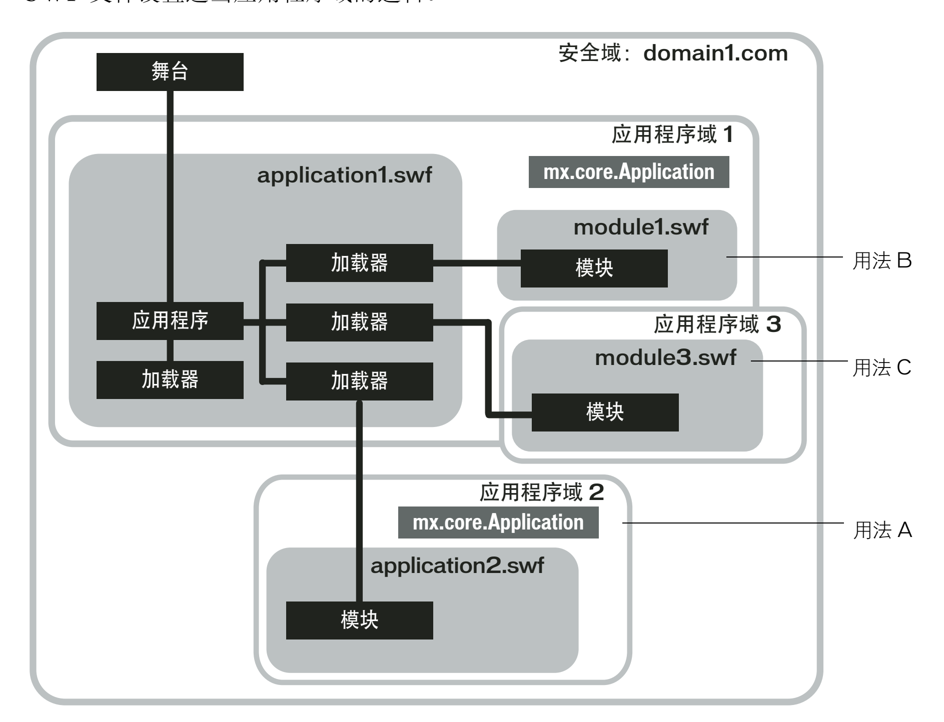
```
  import flash.net.URLRequest;
  import flash.system.ApplicationDomain;
  import flash.system.LoaderContext;

  public class ApplicationDomainExample extends Sprite
  {
    private var ldr:Loader;
    public function ApplicationDomainExample()
    {
      ldr = new Loader();
      var req:URLRequest = new URLRequest("Greeter.swf");
      var ldrContext:LoaderContext = new LoaderContext(false,
  ApplicationDomain.currentDomain);
      ldr.contentLoaderInfo.addEventListener(Event.COMPLETE,
  completeHandler);
      ldr.load(req, ldrContext);
    }
    private function completeHandler(event:Event):void
    {
      ApplicationDomain.currentDomain.getDefinition("Greeter");
      var myGreeter:Greeter = Greeter(event.target.content);
      var message:String = myGreeter.welcome("Tommy");
      trace(message); // Hello, Tommy
    }
  }
}
```

使用应用程序域时，还要记住以下几点：

- SWF 文件中的所有代码被定义为存在于应用程序域中。主应用程序在“当前域”中运行。“系统域”中包含所有应用程序域（包括当前域），也就是，它包含所有 Flash Player 类。
- 所有应用程序域（除系统域外）都有关联的父域。主应用程序的应用程序域的父域是系统域。已加载的类仅在其父级中没有相关定义时才进行定义。不能用较新的定义覆盖已加载类的定义。

下图显示了某个应用程序在单个域 (domain1.com) 中加载多个 SWF 文件的内容。根据加载内容的不同，可以使用不同的应用程序域。紧跟的文本说明用于为应用程序中的每个 SWF 文件设置适当应用程序域的逻辑。




主应用程序文件为 application1.swf。它包含从其它 SWF 文件加载内容的 Loader 对象。在此方案下，当前域为 Application domain 1。用法 A、用法 B 和用法 C 说明了为应用程序中的每个 SWF 文件设置适当应用程序域的不同方法。

**用法 A**：通过创建系统域的子级划分子级 SWF 文件。在示意图中，Application domain 2 创建为系统域的子级。 application2.swf 文件在 Application domain 2 中加载，因此其类定义从 application1.swf 中定义的类中划分出来。

此方法的一个用处是使旧版应用程序能够动态加载相同应用程序的更新版本，而不会发生冲突。之所以不发生冲突，是因为尽管使用的是同样的类名称，但它们划分到不同的应用程序域中。

以下代码将创建作为系统域子级的应用程序域：

```
request.url = "application2.swf";
request.applicationDomain = new ApplicationDomain();
```

**用法 B**：在当前类定义中添加新的类定义。module1.swf 的应用程序域设置为当前域 (Application domain 1)。这可让您将新的类定义添加到应用程序的当前一组类定义中。这可用于主应用程序的运行时共享库。加载的 SWF 被视为远程共享库 (RSL)。使用此方法可以在应用程序启动之前使用预加载器加载 RSL。

以下代码将某应用程序域设置为当前域：

```
request.url = "module1.swf";
request.applicationDomain = ApplicationDomain.currentDomain;
```

**用法 C**：通过创建当前域的新子域，使用父级的类定义。module3.swf 的应用程序域是当前域的子级，并且子级使用所有类的父级的版本。此方法的一个用处可能是作为一个使用主应用程序的类型的多屏幕丰富 Internet 应用程序 (RIA) 模块，该模块作为主应用程序的子级加载。如果能够确保所有类始终更新为向后兼容，并且正在加载的应用程序始终比其加载的软件的版本新，则子级将使用父级版本。如果可以确保不继续拥有对子级 SWF 的引用，则拥有了新的应用程序域还使您能够卸载所有的类定义以便于垃圾回收。

此方法使加载的模块可以共享加载者的 singleton 对象和静态类成员。

以下代码将创建当前域的新子域：

```
request.url = "module3.swf";
request.applicationDomain = new 
  ApplicationDomain(ApplicationDomain.currentDomain);
```

# 使用 IME 类

通过使用 IME 类，您可以在 Flash Player 中运用操作系统的 IME。

使用 ActionScript 可以确定以下内容：

- 用户的计算机上是否安装了 IME (`Capabilities.hasIME`)
- 用户计算机上是否启用了 IME (`IME.enabled`)
- 当前 IME 使用的转换模式 (`IME.conversionMode`)

可以使用特定 IME 上下文关联输入文本字段。在输入字段之间切换时，还可以在平假名（日文）、全角数字、半角数字、直接输入等之间切换 IME。

利用 IME，用户可键入多字节语言 （例如中文、日文和韩文）的非 ASCII 文本字符。

有关使用 IME 的详细信息，请参阅要为其开发应用程序的操作系统的文档。要获取其它资源，请参阅以下 Web 站点：

- http://www.microsoft.com/globaldev/default.mspx
- http://developer.apple.com/documentation/
- http://java.sun.com/



如果用户计算机上 IME 未处于活动状态，则调用 IME 方法或属性 （除 `Capabilities.hasIME` 之外）将失败。一旦手动激活 IME，随后对 IME 方法和属性的 ActionScript 调用即会正常运行。例如，如果使用日文 IME，则必须在调用任何 IME 方法或属性之前将它激活。

## 查看是否已安装并启用了 IME

调用任何 IME 方法或属性之前，应始终检查用户计算机上当前是否已安装并启用 IME。以下代码说明了在调用任何方法之前如何检查用户是否安装并启用了 IME：

```
if (Capabilities.hasIME)
{
  if (IME.enabled)
  {
    trace("IME is installed and enabled.");
  }
  else
  {
    trace("IME is installed but not enabled.Please enable your IME and try 
  again.");
  }
}
else
{
  trace("IME is not installed.Please install an IME and try again.");
}
```

上面的代码先使用 `Capabilities.hasIME` 属性检查用户是否安装了 IME。如果将该属性设置为 `true`，则代码使用 `IME.enabled` 属性检查用户当前是否已启用了 IME。

## 确定当前启用的是哪种 IME 转换模式

构建多语言应用程序时，您可能需要确定用户当前启用的是哪种 IME 转换模式。以下代码说明了如何检查用户是否安装了 IME，以及在安装 IME 的情况下当前启用的是哪种 IME 转换模式：

```
if (Capabilities.hasIME)
{
  switch (IME.conversionMode)
  {
    case IMEConversionMode.ALPHANUMERIC_FULL:
      tf.text = "Current conversion mode is alphanumeric (full-width).";
      break;
    case IMEConversionMode.ALPHANUMERIC_HALF:
      tf.text = "Current conversion mode is alphanumeric (half-width).";
      break;
    case IMEConversionMode.CHINESE:
      tf.text = "Current conversion mode is Chinese.";
      break;
    case IMEConversionMode.JAPANESE_HIRAGANA:
      tf.text = "Current conversion mode is Japananese Hiragana.";
      break;
    case IMEConversionMode.JAPANESE_KATAKANA_FULL:
```

```
      tf.text = "Current conversion mode is Japanese Katakana (full-
  width).";
      break;
    case IMEConversionMode.JAPANESE_KATAKANA_HALF:
      tf.text = "Current conversion mode is Japanese Katakana (half-
  width).";
      break;
    case IMEConversionMode.KOREAN:
      tf.text = "Current conversion mode is Korean.";
      break;
    default:
      tf.text = "Current conversion mode is " + IME.conversionMode + ".";
      break;
  }
}
else
{
  tf.text = "Please install an IME and try again.";
}
```

上面的代码先检查用户是否安装了 IME。接下来，该代码通过将 IME.conversionMode 属性与 IMEConversionMode 类中的每个常量进行比较，检查当前 IME 使用的是哪种转换模式。

## 设置 IME 转换模式

更改用户的 IME 的转换模式时，您需要确保将代码封装在 try..catch 块中，因为使用 conversionMode 属性设置转换模式时，如果 IME 不能设置转换模式，则可能会引发错误。以下代码说明了在设置 IME.conversionMode 属性时如何使用 try..catch 块：

```
var statusText:TextField = new TextField;
statusText.autoSize = TextFieldAutoSize.LEFT;
addChild(statusText);
if (Capabilities.hasIME)
{
  try
  {
    IME.enabled = true;
    IME.conversionMode = IMEConversionMode.KOREAN;
    statusText.text = "Conversion mode is " + IME.conversionMode + ".";
  }
  catch (error:Error)
  {
    statusText.text = "Unable to set conversion mode.\n" + error.message;
  }
}
```

上面的代码先创建一个文本字段，该字段用于向用户显示状态消息。接下来，如果已安装 IME，该代码会启用 IME 并将转换模式设置为“韩文”。如果用户计算机上未安装韩文 IME，Flash Player 将引发错误，并由 try..catch 块捕获。try..catch 块会在先前创建的文本字段中显示该错误消息。

## 为特定文本字段禁用 IME

在某些情况下，最好在用户键入字符时禁用其 IME。例如，如果有一个文本字段只接受数字输入，您可能不想让 IME 出现并减缓数据输入的速度。

以下示例说明了如何侦听 `FocusEvent.FOCUS_IN` 和 `FocusEvent.FOCUS_OUT` 事件并相应地禁用用户的 IME：

```
var phoneTxt:TextField = new TextField();
var nameTxt:TextField = new TextField();

phoneTxt.type = TextFieldType.INPUT;
phoneTxt.addEventListener(FocusEvent.FOCUS_IN, focusInHandler);
phoneTxt.addEventListener(FocusEvent.FOCUS_OUT, focusOutHandler);
phoneTxt.restrict = "0-9";
phoneTxt.width = 100;
phoneTxt.height = 18;
phoneTxt.background = true;
phoneTxt.border = true;
addChild(phoneTxt);

nameField.type = TextFieldType.INPUT;
nameField.x = 120;
nameField.width = 100;
nameField.height = 18;
nameField.background = true;
nameField.border = true;
addChild(nameField);

function focusInHandler(event:FocusEvent):void
{
  if (Capabilities.hasIME)
  {
    IME.enabled = false;
  }
}
function focusOutHandler(event:FocusEvent):void
{
  if (Capabilities.hasIME)
  {
    IME.enabled = true;
  }
}
```

该示例创建两个输入文本字段 `phoneTxt` 和 `nameTxt`，然后在 `phoneTxt` 文本字段中添加两个事件侦听器。当用户将焦点设置为 `phoneTxt` 文本字段时，将调度 `FocusEvent.FOCUS_IN` 事件并禁用 IME。当 `phoneTxt` 文本字段失去焦点时，将调度 `FocusEvent.FOCUS_OUT` 事件以重新启用 IME。

## 侦听 IME 合成事件

设置合成字符串时会调度 IME 合成事件。例如，如果用户启用了 IME 并键入日文字符串，在用户选择合成字符串时，即会调度 `IMEEvent.IME_COMPOSITION` 事件。为了侦听 `IMEEvent.IME_COMPOSITION` 事件，您需要在 System 类的静态 `ime` 属性中添加一个事件侦听器 (`flash.system.System.ime.addEventListener(...)`)，如以下示例所示：

```
var inputTxt:TextField;
var outputTxt:TextField;

inputTxt = new TextField();
inputTxt.type = TextFieldType.INPUT;
inputTxt.width = 200;
inputTxt.height = 18;
inputTxt.border = true;
inputTxt.background = true;
addChild(inputTxt);

outputTxt = new TextField();
outputTxt.autoSize = TextFieldAutoSize.LEFT;
outputTxt.y = 20;
addChild(outputTxt);

if (Capabilities.hasIME)
{
  IME.enabled = true;
  try
  {
    IME.conversionMode = IMEConversionMode.JAPANESE_HIRAGANA;
  }
  catch (error:Error)
  {
    outputTxt.text = "Unable to change IME.";
  }
  System.ime.addEventListener(IMEEvent.IME_COMPOSITION,
  imeCompositionHandler);
}
else
{
  outputTxt.text = "Please install IME and try again.";
}

function imeCompositionHandler(event:IMEEvent):void
{
  outputTxt.text = "you typed: " + event.text;
}
```

上面的代码创建两个文本字段，并将其添加到显示列表中。第一个文本字段 `inputTxt` 是一个输入文本字段，用户可以在其中输入日文文本。第二个文本字段 `outputTxt` 是一个动态文本字段，用于向用户显示错误消息，或回显用户在 `inputTxt` 文本字段中键入的日文字符串。

# 示例：检测系统功能

CapabilitiesExplorer 示例说明如何使用 flash.system.Capabilities 类来确定用户的 Flash Player 支持哪些功能。该示例讲授以下方法：

- 使用 Capabilities 类检测用户的 Flash Player 支持哪些功能
- 使用 ExternalInterface 类检测用户的浏览器支持哪些浏览器设置

要获取该范例的应用程序文件，请访问 www.adobe.com/go/learn_programmingAS3samples_flash_cn。可以在 Samples/CapabilitiesExplorer 文件夹中找到 CapabilitiesExplorer 应用程序文件。该应用程序包含下列文件：

| 文件 | 说明 |
| --- | --- |
| CapabilitiesExplorer.fla<br>或<br>CapabilitiesExplorer.mxml | Flash 或 Flex 中的主应用程序文件 （分别为 FLA 和 MXML）。 |
| com/example/programmingas3/capabilities/<br>CapabilitiesGrabber.as | 用于提供应用程序的主要功能的类，这些主要功能包括在数组中添加系统功能、对项目进行排序以及使用 ExternalInterface 类检索浏览器功能。 |
| capabilities.html | 包含与外部 API 进行通信所必需的 JavaScript 的 HTML 容器。 |

## CapabilitiesExplorer 概述

CapabilitiesExplorer.mxml 文件负责设置 CapabilitiesExplorer 应用程序的用户界面。用户的 Flash Player 功能将显示在舞台上的 DataGrid 组件实例中。如果用户是从 HTML 容器运行应用程序且外部 API 可用，也将显示其浏览器功能。

调度主应用程序文件的 `creationComplete` 事件时，将调用 `initApp()` 方法。`initApp()` 方法从 com.example.programmingas3.capabilities.CapabilitiesGrabber 类中调用 `getCapabilities()` 方法。`initApp()` 方法的代码如下所示：

```
private function initApp():void
{
  var dp:Array = CapabilitiesGrabber.getCapabilities();
  capabilitiesGrid.dataProvider = dp;
}
```

`CapabilitiesGrabber.getCapabilities()` 方法返回排序的 Flash Player 和浏览器功能的数组，然后将其设置为舞台上的 `capabilitiesGrid` DataGrid 组件实例的 `dataProvider` 属性。

## CapabilitiesGrabber 类概述

CapabilitiesGrabber 类的静态 `getCapabilities()` 方法将 flash.system.Capabilities 类中的每个属性添加到数组 (`capDP`)。然后调用 CapabilitiesGrabber 类中的静态 `getBrowserObjects()` 方法。`getBrowserObjects()` 方法使用外部 API 循环访问浏览器的导航器对象 （包含浏览器的功能）。`getCapabilities()` 方法如下所示：

```
public static function getCapabilities():Array
{
  var capDP:Array = new Array();
  capDP.push({name:"Capabilities.avHardwareDisable",
  value:Capabilities.avHardwareDisable});
  capDP.push({name:"Capabilities.hasAccessibility",
  value:Capabilities.hasAccessibility});
  capDP.push({name:"Capabilities.hasAudio", value:Capabilities.hasAudio});
  ...
  capDP.push({name:"Capabilities.version", value:Capabilities.version});
  var navArr:Array = CapabilitiesGrabber.getBrowserObjects();
  if (navArr.length > 0)
  {
    capDP = capDP.concat(navArr);
  }
  capDP.sortOn("name", Array.CASEINSENSITIVE);
  return capDP;
}
```

`getBrowserObjects()` 方法返回一个数组，它包含浏览器的导航器对象中的每个属性。如果该数组的长度为一个或多个项目，则将浏览器功能的数组 (`navArr`) 追加到 Flash Player 功能的数组 (`capDP`) 末尾，并按字母顺序对整个数组进行排序。最后，排序后的数组将返回到主应用程序文件，随后填充数据网格。`getBrowserObjects()` 方法的代码如下所示：

```
private static function getBrowserObjects():Array
{
  var itemArr:Array = new Array();
  var itemVars:URLVariables;
  if (ExternalInterface.available)
  {
    try
    {
      var tempStr:String = ExternalInterface.call("JS_getBrowserObjects");
      itemVars = new URLVariables(tempStr);
      for (var i:String in itemVars)
      {
        itemArr.push({name:i, value:itemVars[i]});
      }
    }
    catch (error:SecurityError)
    {
      // ignore
    }
```

```
    }
    return itemArr;
}
```

如果当前用户环境中具有外部 API， Flash Player 将调用 JavaScript `JS_getBrowserObjects()` 方法，该方法循环访问浏览器的导航器对象并向 ActionScript 返回一个 URL 编码的值的字符串。该字符串随后将转换为 URLVariables 对象 (`itemVars`) 并添加到 `itemArr` 数组中，将向调用脚本返回该数组。

## 与 JavaScript 进行通信

构建 CapabilitiesExplorer 应用程序的最后一部分是编写循环访问浏览器的导航器对象中每个项目所必需的 JavaScript，并将一个名称 - 值对追加到临时数组末尾。container.html 文件中的 JavaScript `JS_getBrowserObjects()` 方法的代码如下所示：

```
<script language="JavaScript">
    function JS_getBrowserObjects()
    {
        // 创建数组以保存浏览器中的每个项目。
        var tempArr = new Array();

        // 循环访问浏览器的导航器对象中的每个项目。
        for (var name in navigator)
        {
            var value = navigator[name];

            // 如果当前值为字符串或 Boolean 对象，则将其添加到
            // 数组中，否则忽略该项目。
            switch (typeof(value))
            {
                case "string":
                case "boolean":

                    // 创建一个将添加到数组中的临时字符串。
                    // 确保使用 JavaScript 的
                    // escape() 函数对值进行 URL 编码。
                    var tempStr = "navigator." + name + "=" + escape(value);
                    // 将 URL 编码的名称 / 值对移到数组上。
                    tempArr.push(tempStr);
                    break;
            }
        }
        // 循环访问浏览器的屏幕对象中的每个项目。
        for (var name in screen)
        {
            var value = screen[name];

            // 如果当前值为数字，则将其添加到数组中，否则
```

```
        // 忽略该项目。
        switch (typeof(value))
        {
          case "number":
            var tempStr = "screen." + name + "=" + escape(value);
            tempArr.push(tempStr);
            break;
        }
      }
      // 以 URL 编码的名称 - 值对字符串的形式返回该数组。
      return tempArr.join("&");
    }
</script>
```

代码先创建一个临时数组，该数组用于保存导航器对象中的所有名称 - 值对。接下来，使用 `for..in` 循环对导航器对象进行循环访问，并求出当前值的数据类型以过滤掉不需要的值。在此应用程序中，我们只需要 String 或 Boolean 值，其它数据类型（例如函数或数组）将被忽略。导航器对象中的每个 String 或 Boolean 值将追加到 `tempArr` 数组末尾。接下来，使用 `for..in` 循环对浏览器的屏幕对象进行循环访问，将每个数值添加到 `tempArr` 数组中。最后，使用 `Array.join()` 方法将临时数组转换为字符串。该数组使用 (&) 符号作为分隔符，这使得 ActionScript 可以使用 URLVariables 类轻松分析数据。

第 24 章

# 打印

24

Adobe® Flash® Player 9 可以与操作系统的打印接口通信，以便您可以将页面传递给打印后台处理程序。 Flash Player 发送到后台处理程序的每个页面可能包含可见的、动态的或屏幕上未显示给用户的内容，其中包括数据库值和动态文本。另外， Flash Player 根据用户的打印机设置来设置 flash.printing.PrintJob 类的属性，以便您可以适当地设置页面格式。

本章详细介绍了使用 flash.printing.PrintJob 类方法和属性创建打印作业、读取用户的打印设置以及根据 Flash Player 和用户操作系统的反馈调整打印作业的策略。

## 目录

# 打印基础知识

## 打印简介

在 ActionScript 3.0 中，可以使用 PrintJob 类来创建显示内容的快照以转换为打印输出中的墨水和纸张表示形式。在某些方面，设置要打印的内容与设置在屏幕上显示的内容是相同的；即可以放置元素和调整其大小以创建所需的布局。但是，打印具有某些特性，而使其不同于屏幕布局。例如，打印机使用的分辨率不同于计算机显示器；计算机屏幕的内容是动态的并且可能会发生变化，而打印的内容本身静态的；在准备进行打印时，您需要考虑固定纸张大小的限制以及多页打印的可能性。

即使这些差异看起来是显而易见的，但在使用 ActionScript 设置打印时一定要记住这些不同之处。由于精确打印取决于您指定的值和用户打印机特性的组合，因此，您可以使用 PrintJob 类中包含的属性来确定需要考虑的重要用户打印机特性。

## 常见打印任务

本章介绍了以下常见的打印任务：

- 启动打印作业
- 在打印作业中添加页面
- 确定用户是否取消了打印作业
- 指定是使用位图还是矢量呈现
- 设置页面大小、缩放和方向
- 指定可打印的内容区域
- 将屏幕大小转换为页面大小
- 打印多页打印作业

## 重要概念和术语

以下参考列表包含将会在本章中使用的重要术语：

- 后台处理程序 (Spooler)：操作系统或打印机驱动程序软件的一部分，用于在等待打印页面时跟踪页面，并在打印机可用时将其发送到打印机。
- 页面方向 (Page orientation)：打印的内容相对于纸张的旋转角度：水平（横向）或垂直（纵向）。
- 打印作业 (Print job)：组成单个打印输出的页面或页面集。

## 完成本章中的示例

学习本章的过程中，您可能想要测试示例代码清单。本章中的许多代码清单是较小的代码部分，而不是完整的打印工作示例或用于检查值的完整代码。测试这些示例涉及创建要打印的元素以及将代码清单与这些元素一起使用。本章中的最后两个示例是完整的打印示例；这些示例包含用于定义要打印的内容的代码，以及执行打印任务的代码。

要测试示例代码清单，请执行以下操作：

1. 创建一个新的 Flash 文档。
2. 选择时间轴的第 1 帧中的关键帧，并打开 “动作”面板。
3. 将代码清单复制到 “脚本”窗格中。
4. 从主菜单中，选择 “控制” > “测试影片”以创建 SWF 文件并测试该示例。

# 打印页面

使用 PrintJob 类的实例来处理打印。要通过 Flash Player 打印基本页面，请依次使用下面四个语句：

- `new PrintJob()`：创建指定的打印作业名称的新实例。
- `PrintJob.start()`：为操作系统启动打印过程 （系统将为用户调用打印对话框），并填充打印作业的只读属性。
- `PrintJob.addPage()`：包含有关打印作业内容的详细信息，其中包括 Sprite 对象 （及其包含的任何子级） 、打印区域的大小以及打印机应将图像打印为矢量图形还是位图图像。您可以使用对 `addPage()` 的连续调用，在多个页面上打印多个 sprite。
- `PrintJob.send()`：将页面发送到操作系统的打印机。

例如，一个非常简单的打印作业脚本如下所示（包括用于编译的 `package`、`import` 和 `class` 语句）：

```
package
{
  import flash.printing.PrintJob;
  import flash.display.Sprite;

  public class BasicPrintExample extends Sprite
  {
    var myPrintJob:PrintJob = new PrintJob();
    var mySprite:Sprite = new Sprite();

    public function BasicPrintExample()
    {
      myPrintJob.start();
      myPrintJob.addPage(mySprite);
      myPrintJob.send();
```

```
      }
    }
}
```

| 注意 | 此示例是为了说明打印作业脚本的基本元素，不包含任何错误处理。要生成脚本以正确响应用户取消打印作业的操作，请参阅第 618 页的 "处理异常和返回值"。 |
|---|---|

如果您出于任何原因而需要清除 PrintJob 对象的属性，请将 PrintJob 变量设置为 `null`（如 `myPrintJob = null` 中所示）。

# Flash Player 任务和系统打印

由于 Flash Player 向操作系统的打印接口调度页面，因此您应了解 Flash Player 和操作系统自身的打印接口所管理的任务范围。Flash Player 可启动打印作业，读取打印机的部分页面设置，将打印作业的内容传递给操作系统，并验证用户或系统是否已取消打印作业。其它过程（例如显示特定于打印机的对话框、取消后台打印作业或报告打印机的状态）都由操作系统处理。如果出现打印作业启动或格式设置问题，Flash Player 能够做出响应，但只能报告操作系统打印接口的某些属性或条件。作为开发人员，您编写的代码应能响应这些属性或条件。

## 处理异常和返回值

您应检查在用户已取消打印作业的情况下，在执行 `addPage()` 和 `send()` 调用之前 `PrintJob.start()` 方法是否返回 `true`。一种在继续之前检查是否已取消这些方法的简单途径是，将它们包含在 `if` 语句中，如下所示：

```
if (myPrintJob.start())
{
    // 此处为 addPage() 和 send() 语句
}
```

如果 `PrintJob.start()` 为 `true`，即表示用户已选择 Print（或者 Flash Player 已启用 Print 命令），则可以调用 `addPage()` 和 `send()` 方法。

另外，为了帮助管理打印过程， Flash Player 现在对 `PrintJob.addPage()` 方法引发异常，以便您可以捕获错误并向用户提供信息和选项。如果 `PrintJob.addPage()` 方法失败，则可以调用其它函数或停止当前打印作业。您可以通过将 `addPage()` 调用嵌入 `try..catch` 语句来捕获这些异常，如下例所示。在该例中，`[params]` 是指定实际打印内容的参数的占位符：

```
if (myPrintJob.start())
{
    try
    {
        myPrintJob.addPage([params]);
    }
    catch (error:Error)
```

```
  {
    // 处理错误,
  }
  myPrintJob.send();
}
```

打印作业启动后，您就可以使用 PrintJob.addPage() 添加内容，并查看是否生成异常（例如，如果用户已取消打印作业）。如果生成异常，则可以向 catch 语句添加逻辑，以便向用户（或 Flash Player）提供信息和选项，或者可以停止当前打印作业。如果成功添加了页面，则可以继续使用 PrintJob.send() 将页面发送到打印机。

如果 Flash Player 在将打印作业发送到打印机时遇到问题（例如，如果打印机处于脱机状态），则也可以捕获该异常并向用户（或 Flash Player）提供信息或更多选项（例如显示消息文本或在 Flash 动画中提供警告）。例如，您可以为 if..else 语句中的文本字段分配新文本，如以下代码所示：

```
if (myPrintJob.start())
{
  try
  {
    myPrintJob.addPage([params]);
  }
  catch (error:Error)
  {
    // 处理错误。
  }
  myPrintJob.send();
}
else
{
  myAlert.text = "Print job canceled";
}
```

有关运行正常的示例，请参阅第 625 页的 “示例：缩放、裁剪和拼接”。

## 处理页面属性

用户在 “打印”对话框中单击 “确定”且 PrintJob.start() 返回 true 后，您就可以访问由打印机设置定义的属性。这些设置包括纸张宽度、纸张高度（pageHeight 和 pageWidth）以及纸张上内容的方向。由于这些设置属于打印机设置，不受 Flash Player 控制，因此您无法更改这些设置；但是，您可以使用它们将发送到打印机的内容对齐以与当前设置匹配。有关详细信息，请参阅第 621 页的 “设置大小、缩放和方向”。

## 设置矢量或位图呈现

您可以将打印作业手动设置为将每个页面在后台处理为矢量图形或位图图像。在某些情况下，矢量打印相对于位图打印而言会生成更小的后台文件和效果更佳的图像。但是，如果您的内容包含位图图像，并且您要保留任何 Alpha 透明度或色彩效果，则应将页面打印为位图图像。另外，非 PostScript 打印机会自动将任何矢量图形转换为位图图像。您可以在 `PrintJob.addPage()` 的第三个参数中指定位图打印，方法是传递一个 PrintJobOptions 对象并将 `printAsBitmap` 参数设置为 `true`，如下所示：

```
var options:PrintJobOptions = new PrintJobOptions();
options.printAsBitmap = true;
myPrintJob.addPage(mySprite, null, options);
```

如果没有指定第三个参数的值，打印作业将使用默认设置，即矢量打印。



如果您不想指定 `printArea` （第二个参数）的值，但想指定位图打印的值，请为 `printArea` 使用 `null`。

## 为打印作业语句定时

与 ActionScript 的先前版本不同，ActionScript 3.0 未将 PrintJob 对象限定在单帧。然而，由于在用户单击 "打印" 对话框中的 "确定" 按钮之后，操作系统会向用户显示打印状态信息，所以应尽快调用 `PrintJob.addPage()` 和 `PrintJob.send()`，以将页面发送到后台处理程序。如果到达包含 `PrintJob.send()` 调用的帧时发生延迟，将会延迟打印过程。

在 ActionScript 3.0 中，脚本超时限制为 15 秒。因此，打印作业序列中各个主要语句间的时间间隔不能超过 15 秒。也就是说， 15 秒脚本超时限制适用于以下时间间隔：

- `PrintJob.start()` 和第一个 `PrintJob.addPage()` 之间
- `PrintJob.addPage()` 和下一个 `PrintJob.addPage()` 之间
- 最后一个 `PrintJob.addPage()` 和 `PrintJob.send()` 之间

如果以上任何一个间隔时间超过了 15 秒，则对 PrintJob 实例的下一次 `PrintJob.start()` 调用将返回 `false`，并且对 PrintJob 实例的下一个 `PrintJob.addPage()` 将使 Flash Player 引发运行时异常。

# 设置大小、缩放和方向

第 617 页的“打印页面”一节详细介绍了基本打印作业的步骤，此作业的输出直接反映了与指定的 sprite 的屏幕大小和位置等效的打印。然而，打印机使用不同的分辨率进行打印，并且可以具有对打印 sprite 的外观有不利影响的设置。

Flash Player 可以读取操作系统的打印设置，但请注意，这些属性是只读的：虽然可以响应它们的值，但无法进行设置。例如，您可以查明打印机的页面大小设置，并调整内容以适当该大小。您还可以确定打印机的边距设置和页面方向。为响应打印机设置，您最好指定打印区域，调整屏幕的分辨率与打印机的磅度量单位之间的差异，或转换内容以符合用户打印机的大小或方向设置。

## 为打印区域使用矩形

使用 `PrintJob.addPage()` 方法可以指定要打印的 sprite 的区域。第二个参数 `printArea` 为 Rectangle 对象的形式。您可以选择三种方法来提供该参数的值：

- 创建一个具有特定属性的 Rectangle 对象，然后将该矩形用于 `addPage()` 调用，如下例所示：

```
private var rect1:Rectangle = new Rectangle(0, 0, 400, 200);
myPrintJob.addPage(sheet, rect1);
```

- 如果您尚未指定 Rectangle 对象，则可以在该调用本身内指定，如下例所示：

```
myPrintJob.addPage(sheet, new Rectangle(0, 0, 100, 100));
```

- 如果您打算为 `addPage()` 调用中的第三个参数提供值，但不想指定矩形，则可以对第二个参数使用 `null`，如下所示：

```
myPrintJob.addPage(sheet, null, options);
```



如果您打算为打印尺寸指定矩形，请记住要导入 `flash.display.Rectangle` 类。

## 比较磅和像素

矩形的宽度和高度以像素为单位。打印机使用磅来作为打印的度量单位。磅的实际大小是固定的（1/72 英寸），但是在屏幕上，像素的大小取决于特定屏幕的分辨率。像素和磅之间的转换比率取决于打印机设置以及 sprite 是否经过缩放。一个 72 个像素宽的 sprite 在未经缩放的情况下打印输出将为一英寸宽，这时，一磅等于一个像素，且与屏幕分辨率无关。

您可以使用以下换算公式将英寸或厘米转换为缇或磅 （1 缇为 1/20 磅）：

- 1 磅 = 1/72 英寸 = 20 缇
- 1 英寸 = 72 磅 = 1440 缇
- 1 厘米 = 567 缇

如果省略了 `printArea` 参数或错误地传递了该参数，将打印 sprite 的整个区域。

## 缩放

如果要在打印前对 Sprite 对象进行缩放，请在调用 `PrintJob.addPage()` 方法之前设置缩放属性（请参阅第 346 页的 "处理大小和缩放对象"），并在打印后将它们重新设置为原始值。 Sprite 对象的缩放与 `printArea` 属性无关。也就是说，如果指定一个 50 x 50 个像素的打印区域，则会打印 2500 个像素。如果对 Sprite 对象进行缩放，则同样会打印 2500 个像素，但按缩放后的大小打印 Sprite 对象。

有关示例，请参阅第 625 页的 "示例：缩放、裁剪和拼接"。

## 横向或纵向打印

由于 Flash Player 可以检测方向设置，因此，您可以在 ActionScript 中构建逻辑来调整内容大小或旋转以响应打印机设置，如下例所示：

```
if (myPrintJob.orientation == PrintJobOrientation.LANDSCAPE)
{
   mySprite.rotation = 90;
}
```



如果您打算读取纸张上内容方向的系统设置，请记住要按照以下准则导入 PrintJobOrientation 类：

```
import flash.printing.PrintJobOrientation;
```

PrintJobOrientation 类提供了定义页面上的内容方向的常量值。

## 响应页面高度和宽度

使用类似于处理打印机方向设置的策略，可以读取页面高度和宽度设置，并通过将某些逻辑嵌入 `if` 语句中来响应这些设置。下面的代码显示了一个示例：

```
if (mySprite.height > myPrintJob.pageHeight)
{
   mySprite.scaleY = .75;
}
```

此外，还可以通过比较页面尺寸和纸张尺寸来确定页面的边距设置，如下例所示：

```
margin_height = (myPrintJob.paperHeight - myPrintJob.pageHeight) / 2;
margin_width = (myPrintJob.paperWidth - myPrintJob.pageWidth) / 2;
```

# 示例：多页打印

打印多页内容时，可以将每一页内容与不同的 sprite（本例中为 `sheet1` 和 `sheet2`）关联，然后为每个 sprite 使用 `PrintJob.addPage()`。以下代码说明了这种方法：

```
package
{
  import flash.display.MovieClip;
  import flash.printing.PrintJob;
  import flash.printing.PrintJobOrientation;
  import flash.display.Stage;
  import flash.display.Sprite;
  import flash.text.TextField;
  import flash.geom.Rectangle;

  public class PrintMultiplePages extends MovieClip
  {
    private var sheet1:Sprite;
    private var sheet2:Sprite;

    public function PrintMultiplePages():void
    {
      init();
      printPages();
    }

    private function init():void
    {
      sheet1 = new Sprite();
      createSheet(sheet1, "Once upon a time...", {x:10, y:50, width:80,
  height:130});
      sheet2 = new Sprite();
      createSheet(sheet2, "There was a great story to tell, and it ended
  quickly.\n\nThe end.", null);
    }

    private function createSheet(sheet:Sprite, str:String,
  imgValue:Object):void
    {
      sheet.graphics.beginFill(0xEEEEEE);
      sheet.graphics.lineStyle(1, 0x000000);
      sheet.graphics.drawRect(0, 0, 100, 200);
      sheet.graphics.endFill();

      var txt:TextField = new TextField();
      txt.height = 200;
      txt.width = 100;
      txt.wordWrap = true;
      txt.text = str;
```

```
      if (imgValue != null)
      {
        var img:Sprite = new Sprite();
        img.graphics.beginFill(0xFFFFFF);
        img.graphics.drawRect(imgValue.x, imgValue.y, imgValue.width,
imgValue.height);
        img.graphics.endFill();
        sheet.addChild(img);
      }
      sheet.addChild(txt);
    }

    private function printPages():void
    {
      var pj:PrintJob = new PrintJob();
      var pagesToPrint:uint = 0;
      if (pj.start())
      {
        if (pj.orientation == PrintJobOrientation.LANDSCAPE)
        {
          throw new Error("Page is not set to an orientation of
portrait.");
        }

        sheet1.height = pj.pageHeight;
        sheet1.width = pj.pageWidth;
        sheet2.height = pj.pageHeight;
        sheet2.width = pj.pageWidth;

        try
        {
          pj.addPage(sheet1);
          pagesToPrint++;
        }
        catch (error:Error)
        {
        // 响应错误。
        }

        try
        {
          pj.addPage(sheet2);
          pagesToPrint++;
        }
        catch (error:Error)
        {
          // 响应错误。
        }

        if (pagesToPrint > 0)
```

```
            {
               pj.send();
            }
         }
      }
   }
}
```

# 示例：缩放、裁剪和拼接

在某些情况下，为消除在屏幕上与打印纸张上的显示差异，打印显示对象时您可能需要调整显示对象的大小（或其它属性）。在打印之前调整显示对象的属性（例如使用 scaleX 和 scaleY 属性来调整）时，请注意，如果对象缩放的尺度超出了为打印区域定义的矩形，该对象将被裁剪。页面打印完毕后，您可能还需要重置属性。

以下代码对 txt 显示对象的尺寸进行缩放（但不缩放绿色背景框），结果将按照指定矩形的尺寸对文本字段进行裁剪。打印后，文本字段将返回到在屏幕上显示的原始大小。如果用户从操作系统的“打印”对话框取消打印作业，Flash Player 中的内容将更改以警告用户作业已取消。

```
package
{
   import flash.printing.PrintJob;
   import flash.display.Sprite;
   import flash.text.TextField;
   import flash.display.Stage;
   import flash.geom.Rectangle;

   public class PrintScaleExample extends Sprite
   {
      private var bg:Sprite;
      private var txt:TextField;

      public function PrintScaleExample():void
      {
         init();
         draw();
         printPage();
      }

      private function printPage():void
      {
         var pj:PrintJob = new PrintJob();
         txt.scaleX = 3;
         txt.scaleY = 2;
         if (pj.start())
         {
            trace(">> pj.orientation: " + pj.orientation);
```

```
            trace(">> pj.pageWidth: " + pj.pageWidth);
            trace(">> pj.pageHeight: " + pj.pageHeight);
            trace(">> pj.paperWidth: " + pj.paperWidth);
            trace(">> pj.paperHeight: " + pj.paperHeight);

            try
            {
                pj.addPage(this, new Rectangle(0, 0, 100, 100));
            }
            catch (error:Error)
            {
                // 不执行任何操作。
            }
            pj.send();
        }
        else
        {
            txt.text = "Print job canceled";
        }
        // 重置 txt 缩放属性。
        txt.scaleX = 1;
        txt.scaleY = 1;
    }

    private function init():void
    {
        bg = new Sprite();
        bg.graphics.beginFill(0x00FF00);
        bg.graphics.drawRect(0, 0, 100, 200);
        bg.graphics.endFill();

        txt = new TextField();
        txt.border = true;
        txt.text = "Hello World";
    }

    private function draw():void
    {
        addChild(bg);
        addChild(txt);
        txt.x = 50;
        txt.y = 50;
    }
  }
}
```

第 25 章

# 使用外部 API

25

ActionScript 3.0 外部 API 可实现 ActionScript 与在其中运行 Adobe Flash Player 9 的容器应用程序之间的直接通信。在某些情况下，您可能需要使用外部 API，例如：创建 SWF 文档与 HTML 页中的 JavaScript 之间的交互，或者构建台式机应用程序以使用 Flash Player 来播放 SWF 文件。

本章介绍了如何使用外部 API 与容器应用程序进行交互，如何在 ActionScript 与 HTML 页中的 JavaScript 之间传递数据，以及如何在 ActionScript 和台式机应用程序之间建立通信并交换数据。



本章仅讨论 SWF 中的 ActionScript 与容器应用程序之间的通信，该应用程序包含对加载 SWF 的 Flash Player 实例的引用。应用程序中其它任何使用 Flash Player 的情况都不在本文档的讨论范围之内。 Flash Player 用作浏览器插件或放映文件 （独立应用程序）。可能还在一定程度上支持其它使用方案。

## 目录

# 使用外部 API 的基础知识

## 使用外部 API 简介

虽然在某些情况下可以独立运行 SWF 文件（例如，如果创建 SWF 放映文件），但在大多数情况下， SWF 应用程序作为另一个应用程序中的元素来运行。通常，包含 SWF 的容器是 HTML 文件；在少数情况下， SWF 文件用于台式机应用程序的全部或部分用户界面。

当处理更高级的应用程序时，您可能会发现，需要在 SWF 文件和容器应用程序之间建立通信。例如，网页通常使用 HTML 格式来显示文本或其它信息，并包含 SWF 文件以显示动态可视内容，如图表或视频。在这种情况下，您可能需要建立此类通信，以便当用户单击网页上的按钮时，它将更改 SWF 文件中的某些内容。ActionScript 包含一种称为外部 API 的机制，它有助于在 SWF 文件中的 ActionScript 与容器应用程序中的其它代码之间建立这种类型的通信。

## 常见外部 API 任务

本章介绍了以下常见的外部 API 任务：

- 获取有关容器应用程序的信息
- 使用 ActionScript 调用容器应用程序中的代码，其中包括网页或台式机应用程序
- 从容器应用程序的代码中调用 ActionScript 代码
- 创建代理以简化从容器应用程序调用 ActionScript 代码的过程

## 重要概念和术语

以下参考列表包含将会在本章中使用的重要术语：

- ActiveX 容器 (ActiveX container)：容器应用程序（不是 Web 浏览器），它包含 Flash Player ActiveX 控件的实例以便在应用程序中显示 SWF 内容。
- 容器应用程序 (Container application)：Flash Player 在其中运行 SWF 文件的应用程序，如 Web 浏览器和包含 Flash Player 内容的 HTML 页。
- 放映文件 (Projector)：已转换为独立可执行文件的 SWF 文件，其中包括 Flash Player 以及 SWF 文件的内容。可以使用 Adobe Flash CS3 Professional 或独立的 Flash Player 来创建放映文件。放映文件通常用于以 CD-ROM 形式分发 SWF 文件，或在以下类似情况下进行 SWF 文件分发：下载文件大小不是问题，并且 SWF 作者希望确保用户能够运行 SWF 文件，而无论用户计算机上是否安装了 Flash Player。

- 代理 (Proxy): 这是一个中间应用程序或代码，它代表一个应用程序（“调用应用程序”）调用另一个应用程序（“外部应用程序”）中的代码，并将值返回到调用应用程序。可以出于各种不同的原因来使用代理，其中包括：
  - 通过将调用应用程序中的本机函数调用转换为外部应用程序所理解的格式，简化进行外部函数调用的过程
  - 解决禁止调用方直接与外部应用程序进行通信的安全或其它限制问题
- 序列化 (Serialize): 将对象或数据值转换为某种格式，这种格式可用于通过消息在两个编程系统之间传递值，如通过 Internet 或在一台计算机上运行的两个不同应用程序之间进行传递。

## 完成本章中的示例

学习本章的过程中，您可能想要测试示例代码清单。本章中的许多代码清单是用于演示的较小的代码清单，而不是完整的工作示例或用于检查值的完整代码。由于使用外部 API 需要（根据定义）编写 ActionScript 代码以及容器应用程序中的代码，因此测试示例涉及创建一个容器 （例如，包含 SWF 的网页）和使用代码清单与该容器交互。

**要测试 ActionScript 与 JavaScript 之间的通信的示例，请执行以下操作：**

1. 创建一个新的 Flash 文档并将它保存到您的计算机上。
2. 从主菜单中，选择 “文件” > “发布设置”。
3. 在 “发布设置”对话框中，在 “格式”选项卡上确认选中了 “Flash”和 “HTML”复选框。
4. 单击 “发布”按钮。这将在同一个文件夹中生成一个 SWF 文件和一个 HTML 文件，且两个文件的名称与您用于保存 Flash 文档的名称相同。单击 “确定”关闭 “发布设置”对话框。
5. 取消选择 “HTML”复选框。现在 HTML 页即生成，您将修改该页来添加适当的 JavaScript 代码。取消选择 “HTML”复选框可确保在您修改 HTML 页之后， Flash 在发布 SWF 文件时不会使用新的 HTML 页覆盖您所做的更改。
6. 单击 “确定”关闭 “发布设置”对话框。
7. 使用 HTML 或文本编辑器应用程序打开 Flash 在您发布 SWF 时创建的 HTML 文件。在 HTML 源代码中，添加一个脚本元素，并将示例代码清单中的 JavaScript 代码复制到其中。
8. 保存该 HTML 文件并返回到 Flash。
9. 选择时间轴的第 1 帧中的关键帧，并打开 “动作”面板。
10. 将 ActionScript 代码清单复制到 “脚本”窗格中。

11. 从主菜单中，选择 “文件” > “发布”以使用您所做的更改更新该 SWF 文件。
12. 使用 Web 浏览器打开您编辑过的 HTML 页，查看该页并测试 ActionScript 与 HTML 页之间的通信。

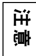
**要测试 ActionScript 与 ActiveX 容器之间的通信的示例，请执行以下操作：**

1. 创建一个新的 Flash 文档并将它保存到您的计算机上。您可能想要将它保存在容器应用程序预计能够在其中找到 SWF 文件的文件夹中。
2. 从主菜单中，选择 “文件” > “发布设置”。
3. 在 “发布设置”对话框中，在 “格式”选项卡上确认仅选中了 “Flash”复选框。
4. 在 “Flash”复选框旁边的 “文件”字段中，单击文件夹图标以选择 SWF 文件将发布到的文件夹。通过设置 SWF 文件的位置，您可以 （举例而言）将 Flash 文档放在一个文件夹中，而将发布的 SWF 文件放在另一个文件夹 （例如包含容器应用程序的源代码的文件夹）中。
5. 选择时间轴的第 1 帧中的关键帧，并打开 “动作”面板。
6. 将示例的 ActionScript 代码复制到 “脚本”窗格中。
7. 从主菜单中，选择 “文件” > “刷新”以重新发布该 SWF 文件。
8. 创建并运行您的容器应用程序，以测试 ActionScript 与容器应用程序之间的通信。

本章末尾的两个示例分别是一个使用外部 API 与 HTML 页通信的完整示例，和一个使用外部 API 与 C# 台式机应用程序通信的完整示例。这些示例包含完整代码，其中包括 ActionScript 和容器错误检查代码，您在使用外部 API 编写代码时将用到这些代码。有关使用外部 API 的另一个完整示例，请参阅 《ActionScript 3.0 语言和组件参考》中 ExternalInterface 类的类示例。

# 外部 API 要求和优点

外部 API 是 ActionScript 中的一部分，它为 ActionScript 与作为 Flash Player 容器的外部应用程序 （通常是 Web 浏览器或独立放映文件应用程序）中运行的代码之间进行通信提供了一种机制。在 ActionScript 3.0 中，外部 API 的功能是由 ExternalInterface 类提供的。在 Flash Player 8 之前的 Flash Player 版本中，使用 `fscommand()` 动作与容器应用程序进行通信。ExternalInterface 类替代了 `fscommand()`，是 JavaScript 与 ActionScript 之间的所有通信的推荐使用机制。




如需使用旧的 `fscommand()` 函数 （例如，为了与较早的应用程序保持兼容或与第三方 SWF 容器应用程序或独立的 Flash Player 进行交互），仍可将其作为 flash.system 包中的包级函数来使用。

ExternalInterface 类是一个子系统，通过它可以轻松地实现从 ActionScript 和 Flash Player 到 HTML 页中的 JavaScript 或任何包含 Flash Player 实例的台式机应用程序的通信。

ExternalInterface 类只在以下情况下可用：

- 在所有受支持的 Internet Explorer for Windows 版本（5.0 和更高版本）中
- 在容器应用程序（例如使用 Flash Player ActiveX 控件实例的台式机应用程序）中
- 在支持 NPRuntime 接口的任何浏览器中（当前包括以下浏览器：
  - Firefox 1.0 及更高版本
  - Mozilla 1.7.5 及更高版本
  - Netscape 8.0 及更高版本
  - Safari 1.3 及更高版本

在其它所有情况下（例如，在独立的播放器中运行），`ExternalInterface.available` 属性均返回 `false`。

从 ActionScript 中，可以在 HTML 页上调用 JavaScript 函数。与 `fscommand()` 相比，外部 API 可提供以下改进功能：

- 可以使用任何 JavaScript 函数，而不仅仅是可与 `fscommand()` 函数一起使用的函数。
- 可以传递任意数量的、具有任意名称的参数；而不是仅限于传递一个命令和一个字符串参数。这为外部 API 提供了比 `fscommand()` 大得多的灵活性。
- 可以传递各种数据类型（例如 Boolean、Number 和 String）；不再仅限于 String 参数。
- 可以接收调用值，该值将立即返回到 ActionScript（作为进行的调用的返回值）。

警告

如果为 HTML 页中的 Flash Player 实例指定的名称（`object` 标签的 `id` 属性）包含连字符 (`-`) 或在 JavaScript 中定义为运算符的其它字符（如 `+`、`*`、`/`、`\`、`.` 等），则在 Internet Explorer 中查看容器网页时，将无法从 ActionScript 调用 ExternalInterface。

此外，如果定义 Flash Player 实例的 HTML 标签（`object` 和 `embed` 标签）嵌套在 HTML `form` 标签中，也将无法从 ActionScript 调用 ExternalInterface。

# 使用 ExternalInterface 类

ActionScript 与容器应用程序之间的通信方式有两种：ActionScript 可以调用容器中定义的代码（如 JavaScript 函数），或者容器中的代码可以调用被指定为可调用函数的 ActionScript 函数。在这两种情况下，都可以将信息发送给被调用的代码，而将结果返回给执行调用的代码。

为了便于这种通信，ExternalInterface 类包含了两个静态属性和两个静态方法。这些属性和方法可用于获取有关外部接口连接的信息，从 ActionScript 执行容器中的代码，以及使 ActionScript 函数可供容器调用。

## 获取有关外部容器的信息

ExternalInterface.available 属性指示当前的 Flash Player 是否位于提供外部接口的容器中。如果外部接口可用，则此属性为 true ；否则，为 false。在使用 ExternalInterface 类中的任何其它功能之前，应始终进行检查以确保当前容器支持外部接口通信，如下所示：

```
if (ExternalInterface.available)
{
  // 在此执行 ExternalInterface 方法调用。
}
```



注意

ExternalInterface.available 属性报告当前容器是否为支持 ExternalInterface 连接的容器类型。它不会报告当前浏览器中是否启用了 JavaScript。

通过使用 ExternalInterface.objectID 属性，您可以确定 Flash Player 实例的唯一标识符（具体来说，是指 Internet Explorer 中 object 标签的 id 属性，或者是指使用 NPRuntime 接口的浏览器中 embed 标签的 name 属性）。这个唯一的 ID 代表浏览器中的当前 SWF 文档，并可用于对 SWF 文档进行引用，例如：在容器 HTML 页中调用 JavaScript 函数时进行引用。当 Flash Player 容器不是 Web 浏览器时，此属性为 null。

## 从 ActionScript 中调用外部代码

ExternalInterface.call() 方法执行容器应用程序中的代码。它至少需要一个参数，即包含容器应用程序中要调用函数的名称的字符串。传递给 ExternalInterface.call() 方法的其它任何参数均作为函数调用的参数传递给容器。

```
// 调用外部函数 “addNumbers”
// 传递两个参数并将该函数的结果
// 赋给变量 “result”
var param1:uint = 3;
var param2:uint = 7;
var result:uint = ExternalInterface.call("addNumbers", param1, param2);
```

如果容器为 HTML 页，此方法将调用具有指定名称的 JavaScript 函数，必须在包含 HTML 页中的 script 元素中定义该函数。 JavaScript 函数的返回值被传递回 ActionScript。

```
<script language="JavaScript">
  // 加上两个数字，然后将结果发送回 ActionScript
  function addNumbers(num1, num2)
  {
    return (num1 + num2);
  }
</script>
```

如果容器为其它的 ActiveX 容器，此方法将导致 Flash Player ActiveX 控件调度它的 `FlashCall` 事件。Flash Player 将指定的函数名及所有参数序列化为一个 XML 字符串。容器可以在事件对象的 `request` 属性中访问该信息，并用它来确定如何执行它自己的代码。为了将值返回 ActionScript，容器代码调用 ActiveX 对象的 `SetReturnValue()` 方法，并将结果（序列化为一个 XML 字符串）作为该方法的参数进行传递。有关该通信使用的 XML 格式的详细信息，请参阅第 634 页的 “外部 API 的 XML 格式”。

无论容器为 Web 浏览器还是为其它 ActiveX 容器，只要调用失败或容器方法没有指定返回值，都将返回 `null`。如果包含环境属于调用代码无权访问的安全沙箱，`ExternalInterface.call()` 方法将引发 SecurityError 异常。可以通过在包含环境中为 `allowScriptAccess` 设置合适的值来解决此问题。例如，要在 HTML 页中更改 `allowScriptAccess` 的值，请编辑 `object` 和 `embed` 标签中的相应属性。

## 从容器中调用 ActionScript 代码

容器只能调用函数中的 ActionScript 代码，而不能调用任何其它 ActionScript 代码。要从容器应用程序调用 ActionScript 函数，必须执行两项操作：向 ExternalInterface 类注册函数，然后从容器的代码调用它。

首先，必须注册 ActionScript 函数，指示其应能够为容器所用。使用 `ExternalInterface.addCallback()` 方法，如下所示：

```
function callMe(name:String):String
{
  return "busy signal";
}
ExternalInterface.addCallback("myFunction", callMe);
```

`addCallback()` 方法采用两个参数。第一个参数为 String 类型的函数名，容器将籍此名称得知要调用的函数。第二个参数为容器调用定义的函数名时要执行的实际 ActionScript 函数。由于这些名称是截然不同的，因此可以指定将由容器使用的函数名，即使实际的 ActionScript 函数具有不同的名称。这在函数名未知的情况下特别有用，例如：指定了匿名函数或需要在运行时确定要调用的函数。

一旦向 ExternalInterface 类注册了 ActionScript 函数，容器就可以实际调用该函数。完成该操作的具体方法依容器的类型而定。例如，在 Web 浏览器的 JavaScript 代码中，使用已注册的函数名调用 ActionScript 函数，就像它是 Flash Player 浏览器对象的方法（即，一个表示 `object` 或 `embed` 标签的 JavaScript 对象的方法）。也就是说，将传递参数并返回结果，就如同调用本地函数一样。

```
<script language="JavaScript">
  // callResult gets the value "busy signal"
  var callResult = flashObject.myFunction("my name");
</script>
...
<object id="flashObject"...>
```

```
  ...
  <embed name="flashObject".../>
</object>
```

或者，在运行于台式机应用程序中的 SWF 文件中调用 ActionScript 函数时，必须将已注册的函数名及所有参数序列化为一个 XML 格式的字符串。然后，将该 XML 字符串作为一个参数来调用 ActiveX 控件的 `CallFunction()` 方法，以实际执行该调用。有关该通信使用的 XML 格式的详细信息，请参阅第 634 页的 “外部 API 的 XML 格式”。

不管是哪种情况，ActionScript 函数的返回值都被传递回容器代码，当调用方为浏览器中的 JavaScript 代码时直接作为值返回，而当调用方为 ActiveX 容器时则会序列化为 XML 格式字符串。

# 外部 API 的 XML 格式

ActionScript 与承载 Shockwave Flash ActiveX 控件的应用程序间的通信使用特定的 XML 格式对函数调用和值进行编码。外部 API 使用的 XML 格式分为两种。一种格式用于表示函数调用。另一种格式用于表示各个值；此格式用于函数中的参数及函数返回值。函数调用的 XML 格式用于对 ActionScript 的调用和来自 ActionScript 的调用。对于来自 ActionScript 的函数调用， Flash Player 将 XML 传递给容器；而对于来自容器的调用，Flash Player 需要容器应用程序将向其传递一个此格式的 XML 字符串。下面的 XML 片断说明了一个 XML 格式的函数调用示例：

```
<invoke name="functionName" returntype="xml">
  <arguments>
    ... (individual argument values)
  </arguments>
</invoke>
```

根节点为 `invoke` 节点。它具有两个属性：`name`，指示要调用的函数的名称；以及 `returntype`，总是为 `xml`。如果函数调用包括参数，则 `invoke` 节点具有一个 `arguments` 子节点，该节点的子节点是使用单个值格式（下面将予以说明）进行了格式设置的参数值。

每个值（包括函数参数和函数返回值）均使用一个格式设置方案，除了实际值之外，该方案还包括数据类型信息。下表列出了 ActionScript 类以及用于对该数据类型的值进行编码的 XML 格式：

| ActionScript 类 / 值 | C# 类 / 值 | 格式 | 注释 |
|---|---|---|---|
| null | null | `<null/>` | |
| Boolean `true` | bool true | `<true/>` | |
| Boolean `false` | bool false | `<false/>` | |
| String | string | `<string>string value</string>` | |
| Number、int、uint | single、double、int、uint | `<number>27.5</number>`<br>`<number>-12</number>` | |
| Array（元素可以是混合类型） | 允许混合类型元素的集合，如 ArrayList 或 object[] | `<array>`<br>`  <property id="0">`<br>`    <number>27.5</number>`<br>`  </property>`<br>`  <property id="1">`<br>`    <string>Hello there!</string>`<br>`  </property>`<br>`  ...`<br>`</array>` | `property` 节点定义各个元素，而 `id` 属性为从 0 开始的数值索引。 |
| Object | 含有字符串键和对象值的字典，如具有字符串键的 HashTable | `<object>`<br>`  <property id="name">`<br>`    <string>John Doe</string>`<br>`  </property>`<br>`  <property id="age">`<br>`    <string>33</string>`<br>`  </property>`<br>`  ...`<br>`</object>` | `property` 节点定义各个属性，而 `id` 属性为属性名称（字符串）。 |
| 其它内置或自定义的类 | | `<null/>`<br>或<br>`<object></object>` | ActionScript 将其它对象编码为 null 或空对象。不管是哪种情况，所有属性值都会丢失。 |

> **注意**　该表举例说明了 ActionScript 类，并且还列出了等效 C# 类；但是，外部 API 可用来与支持 ActiveX 控件的任何编程语言或运行时进行通信，而不仅限于 C# 应用程序。

通过将外部 API 用于 ActiveX 容器应用程序来构建您自己的应用程序时，您可能会发现，编写代理以执行将本机函数调用转换为序列化 XML 格式的任务是非常方便的。有关使用 C# 编写的代理类的示例，请参阅第 646 页的 “深入 ExternalInterfaceProxy 类内部”。

# 示例：将外部 API 用于网页容器

本范例应用程序演示实现 ActionScript 与 Web 浏览器中的 JavaScript 间的通信的正确方法，范例的环境为一个允许用户相互对话的即时消息应用程序（因此该应用程序的名称为：Introvert IM）。它使用外部 API 在网页中的 HTML 表单与 SWF 接口之间发送消息。本示例说明的方法包括：

- 通过验证浏览器在建立通信之前已准备就绪，正确地启动通信
- 检查容器是否支持外部 API
- 从 ActionScript 调用 JavaScript 函数，传递参数，并接收响应中的值
- 使 ActionScript 方法可由 JavaScript 调用，并执行这些调用

要获取该范例的应用程序文件，请访问
www.adobe.com/go/learn_programmingAS3samples_flash_cn。Introvert IM 应用程序文件位于 Samples/IntrovertIM_HTML 文件夹中。该应用程序包含以下文件：

| 文件 | 描述 |
| --- | --- |
| IntrovertIMApp.fla<br>或<br>IntrovertIMApp.mxml | Flash 或 Flex 的主应用程序文件 （分别为 FLA 和 MXML）。 |
| com/example/programmingas3/introvertIM/IMManager.as | 建立和管理 ActionScript 与容器间的通信的类。 |
| com/example/programmingas3/introvertIM/IMMessageEvent.as | 自定义事件类型，从容器接收到消息时由 IMManager 类调度。 |
| com/example/programmingas3/introvertIM/IMStatus.as | 枚举，其值表示可在该应用程序中选择的不同可用性状态值。 |
| html-flash/IntrovertIMApp.html<br>或<br>html-template/index.template.html | Flash 应用程序的 HTML 页 (html-flash/IntrovertIMApp.html)，或用于为 Adobe Flex 应用程序创建容器 HTML 页的模板 (html-template/index.template.html)。此文件包含组成该应用程序的容器部分的所有 JavaScript 函数。 |

## 准备 ActionScript 与浏览器间的通信

外部 API 最常见的用途之一就是允许 ActionScript 应用程序与 Web 浏览器进行通信。使用外部 API 时，ActionScript 方法可以调用使用 JavaScript 编写的代码，反之亦然。由于浏览器的复杂性及其内部呈现页的方式，因此根本无法保证 SWF 文档能够在 HTML 页中的第一个 JavaScript 运行之前注册它的回调。出于这个原因，在从 JavaScript 调用 SWF 文档中的函数之前， SWF 文档应总是调用 HTML 页，以通知它 SWF 文档已准备好接受连接。

例如，通过 IMManager 类执行的一系列步骤，Introvert IM 可确定浏览器是否做好了通信的准备，并为通信准备 SWF 文件。第一个步骤（确定浏览器是否已做好通信准备）是在 IMManager 构造函数中执行的，如下所示：

```
public function IMManager(initialStatus:IMStatus)
{
    _status = initialStatus;

    // 检查容器能否使用外部 API。
    if (ExternalInterface.available)
    {
        try
        {
            // 这会调用 isContainerReady() 方法，该方法又调用
            // 容器以查看是否已加载 Flash Player 以及
            // 容器是否已经准备好接收来自 SWF 的调用。
            var containerReady:Boolean = isContainerReady();
            if (containerReady)
            {
                // 如果容器已准备就绪，则注册 SWF 的函数。
                setupCallbacks();
            }
            else
            {
                // 如果容器未准备就绪，则设置一个 Timer 以便
                // 每隔 100 毫秒调用容器一次。在容器发出响应并表明已准备就绪之后，
                // 该计时器将停止。
                var readyTimer:Timer = new Timer(100);
                readyTimer.addEventListener(TimerEvent.TIMER, timerHandler);
                readyTimer.start();
            }
        }
        ...
    }
    else
    {
        trace("External interface is not available for this container.");
    }
}
```

首先，代码使用 ExternalInterface.available 属性检查外部 API 在当前容器中是否可用。如果可用，它将开始建立通信的过程。由于尝试与外部应用程序通信会产生安全异常及其它错误，因此将代码包括在 try 块中（为了简便起见，列表中省略了相应的 catch 块）。

然后，代码调用 isContainerReady() 方法，如下所示：

```
private function isContainerReady():Boolean
{
  var result:Boolean = ExternalInterface.call("isReady");
  return result;
}
```

isContainerReady() 方法又使用 ExternalInterface.call() 方法来调用 JavaScript 函数 isReady()，如下所示：

```
<script language="JavaScript">
<!--
// ------- Private vars -------
var jsReady = false;
...
// ------- 由 ActionScript 调用的函数 -------
// 调用这些函数的目的是检查页面是否已初始化以及 JavaScript 是否可用
function isReady()
{
  return jsReady;
}
...
// 由 <body> 标签的 onload 事件调用
function pageInit()
{
  // 记录 JavaScript 已准备就绪。
  jsReady = true;
}
...
//-->
</script>
```

isReady() 函数仅返回 jsReady 变量的值。该变量最初为 false；一旦触发了网页的 onload 事件，该变量的值即更改为 true。也就是说，如果 ActionScript 在加载页之前调用 isReady() 函数，则 JavaScript 对 ExternalInterface.call("isReady") 返回 false，因此使得 ActionScript isContainerReady() 方法返回 false。加载页之后， JavaScript isReady() 函数将返回 true，从而使得 ActionScript isContainerReady() 方法也返回 true。

回到 IMManager 构造函数中，根据容器的准备情况，将发生两种情况之一。如果 isContainerReady() 返回 true，则代码调用 setupCallbacks() 方法，该方法将完成与 JavaScript 建立通信的过程。另一方面，如果 isContainerReady() 返回 false，实际上会使该过程陷入停顿状态。创建 Timer 对象，并指示其每隔 100 毫秒调用 timerHandler() 方法一次，如下所示：

```
private function timerHandler(event:TimerEvent):void
{
    // 检查容器是否准备就绪。
    var isReady:Boolean = isContainerReady();
    if (isReady)
    {
        // 如果容器已准备就绪，则不再需要进行检查，
        // 因此将停止该计时器。
        Timer(event.target).stop();
        // 设置 ActionScript 方法，
        // 容器将可以调用这些方法。
        setupCallbacks();
    }
}
```

每次调用 timerHandler() 方法时，它都会再次检查 isContainerReady() 方法的结果。初始化容器后，该方法返回 true。然后，代码停止 Timer，并调用 setupCallbacks() 方法来完成与浏览器建立通信的过程。

## 向 JavaScript 公开 ActionScript 方法

如上例所示，一旦代码确定浏览器已就绪，就将调用 setupCallbacks() 方法。此方法准备 ActionScript 以接收来自 JavaScript 的调用，如下所示：

```
private function setupCallbacks():void
{
    // 向容器注册 SWF 客户端函数
    ExternalInterface.addCallback("newMessage", newMessage);
    ExternalInterface.addCallback("getStatus", getStatus);
    // 通知容器 SWF 已为调用做好准备。
    ExternalInterface.call("setSWFIsReady");
}
```

setCallBacks() 方法通过调用 ExternalInterface.addCallback() 来注册两个可从 JavaScript 调用的方法，从而完成与容器进行通信的准备工作。在此代码中，第一个参数与 ActionScript 中方法的名称是相同的，该参数即 JavaScript 所知道的方法名称（newMessage 和 getStatus）。（在本例中，使用不同的名称毫无益处，因此为了简便起见，重复使用相同的名称。）最后，使用 ExternalInterface.call() 方法来调用 JavaScript 函数 setSWFIsReady()，该函数通知容器 ActionScript 函数已注册。

## 从 ActionScript 到浏览器的通信

Introvert IM 应用程序说明一系列在容器页中调用 JavaScript 函数的示例。在最简单的情况下（`setupCallbacks()` 方法的示例），调用 JavaScript 函数 `setSWFIsReady()` 时不传递任何参数也不接收返回值：

```
ExternalInterface.call("setSWFIsReady");
```

在 `isContainerReady()` 方法的另一个示例中， ActionScript 调用 `isReady()` 函数，并接受响应中的一个布尔值：

```
var result:Boolean = ExternalInterface.call("isReady");
```

还可以使用外部 API 将参数传递给 JavaScript 函数。以 IMManager 类的 `sendMessage()` 方法为例，该方法在用户向他（或她）的 "对话伙伴" 发送新消息时调用：

```
public function sendMessage(message:String):void
{
  ExternalInterface.call("newMessage", message);
}
```

再次使用 `ExternalInterface.call()` 调用指定的 JavaScript 函数，通知浏览器有新消息。另外，消息本身也作为另一个参数传递给 `ExternalInterface.call()`，从而作为参数传递给 JavaScript 函数 `newMessage()`。

## 从 JavaScript 调用 ActionScript 代码

通信必须是双向的， Introvert IM 应用程序也不例外。不仅 Flash Player IM 客户端调用 JavaScript 来发送消息，而且 HTML 表单也调用 JavaScript 代码来向 SWF 文件发送消息和请求信息。例如，当 SWF 文件通知容器它已建立了联系并做好了通信的准备时，浏览器执行的第一个操作就是调用 IMManager 类的 `getStatus()` 方法，来检索 SWF IM 客户端的初始用户可用性状态。此操作是在网页中（在 `updateStatus()` 函数中）执行的，如下所示：

```
<script language="JavaScript">
...
function updateStatus()
{
  if (swfReady)
  {
    var currentStatus = getSWF("IntrovertIMApp").getStatus();
    document.forms["imForm"].status.value = currentStatus;
  }
}
...
</script>
```

代码检查 swfReady 变量的值，该变量跟踪 SWF 文件是否已经通知浏览器，它已在 ExternalInterface 类中注册了其方法。如果 SWF 文件已准备好接收通信，则下一行代码 (var currentStatus = ...) 实际调用 IMManager 类中的 getStatus() 方法。该行代码中执行三个操作：

- 调用 getSWF() JavaScript 函数，返回对表示 SWF 文件的 JavaScript 对象的引用。当 HTML 页中存在多个 SWF 文件时，传递给 getSWF() 的参数决定了返回哪一个浏览器对象。传递给该参数的值必须与 object 标签的 id 属性以及用于包括 SWF 文件的 embed 标签的 name 属性相匹配。
- 在使用对 SWF 文件的引用时， getStatus() 方法就像是 SWF 对象的方法一样被调用。在这种情况下，使用函数名 getStatus，因为它是使用 ExternalInterface.addCallback() 注册的 ActionScript 函数名。
- getStatus() ActionScript 方法返回一个值，该值被赋给 currentStatus 变量，该变量作为 status 文本字段的内容 （value 属性）被赋值。

sendMessage() JavaScript 函数说明如何将参数传递给 ActionScript 函数。(sendMessage() 是用户按 HTML 页上的 “发送”按钮时调用的函数。)

```
<script language="JavaScript">
...
function sendMessage(message)
{
  if (swfReady)
  {
    ...
    getSWF("IntrovertIMApp").newMessage(message);
  }
}
...
</script>
```

newMessage() ActionScript 方法需要使用一个参数，因此，将 JavaScript message 变量作为 JavaScript 代码中 newMessage() 方法调用中的参数传递给 ActionScript。

## 检测浏览器类型

由于各种浏览器在访问内容的方式上存在差异，因此必须总是使用 JavaScript 来检测用户正在运行哪种浏览器，并根据特定于浏览器的语法使用窗口对象或文本对象访问影片，如本例中的 getSWF() JavaScript 函数所示：

```
<script language="JavaScript">
...
function getSWF(movieName)
{
  if (navigator.appName.indexOf("Microsoft") != -1)
  {
```

```
    return window[movieName];
  }
  else
  {
    return document[movieName];
  }
}
...
</script>
```

如果脚本不检测用户的浏览器类型，则在 HTML 容器中播放 SWF 文件时，用户可能会遇到意外情况。

# 示例：将外部 API 用于 ActiveX 容器

本示例说明了如何使用外部 API 在 ActionScript 与使用 ActiveX 控件的台式机应用程序之间进行通信。此示例再次使用了 Introvert IM 应用程序（包括 ActionScript 代码乃至相同的 SWF 文件），因此，不再说明外部 API 在 ActionScript 中的用法。熟悉上一个示例将有助于理解本示例。

本示例中的台式机应用程序是通过 Microsoft Visual Studio .NET 使用 C# 语言编写的。本示例的讨论重点是通过 ActiveX 控件使用外部 API 的特定方法。本示例说明如下技术：

- 从承载 Flash Player ActiveX 控件的台式机应用程序调用 ActionScript 函数
- 从 ActionScript 接收函数调用，并在 ActiveX 容器中处理它们
- 使用 proxy 类来隐藏已序列化的 XML 格式的详细信息， Flash Player 使用该格式将消息发送到 ActiveX 容器

要获取该范例的应用程序文件，请访问 www.adobe.com/go/learn_programmingAS3samples_flash_cn。 Introvert IM C# 文件位于 Samples/IntrovertIM_CSharp 文件夹中。该应用程序包含以下文件：

| 文件 | 描述 |
|---|---|
| AppForm.cs | 含有 C# Windows Forms 接口的应用程序主文件。 |
| bin/Debug/IntrovertIMApp.swf | 该应用程序加载的 SWF 文件。 |
| ExternalInterfaceProxy/ExternalInterfaceProxy.cs | 作为 ActiveX 控件的包装以进行 External Interface 通信的类。它为从 ActionScript 进行调用和接收调用提供了机制。 |
| ExternalInterfaceProxy/ExternalInterfaceSerializer.cs | 执行将 Flash Player 的 XML 格式消息转换为 .NET 对象的任务的类。 |
| ExternalInterfaceProxy/ExternalInterfaceEventArgs.cs | 此文件定义两个 C# 类型 （类）： 一个自定义委托类和一个事件参数类，ExternalInterfaceProxy 类用它们来向侦听器通知来自 ActionScript 的函数调用。 |

| 文件 | 描述 |
| --- | --- |
| ExternalInterfaceProxy/ExternalInterfaceCall.cs | 此类是一个值对象，表示从 ActionScript 到 ActiveX 容器的函数调用，具有用于函数名和参数的属性。 |
| bin/Debug/IntrovertIMApp.swf | 该应用程序加载的 SWF 文件。 |
| obj/AxInterop.ShockwaveFlashObjects.dll, obj/Interop.ShockwaveFlashObjects.dll | Visual Studio .NET 创建的包装程序集，需要使用它们从托管代码访问 Flash Player (Adobe Shockwave® Flash) ActiveX 控件。 |

## Introvert IM C# 应用程序概述

本范例应用程序代表两个相互通信的即时消息客户端程序（一个位于 SWF 文件中，而另一个则是使用 Windows Forms 构建的）。用户界面包含一个 Shockwave Flash ActiveX 控件的实例（在其中加载包含 ActionScript IM 客户端的 SWF 文件）。该接口还包含组成 Windows Forms IM 客户端的若干个文本字段：一个用于输入消息 (`MessageText`)，一个用于显示在客户端之间发送的消息的文本 (`Transcript`)，还有一个用于显示 SWF IM 客户端中设置的可用性状态 (`Status`)。

## 包含 Shockwave Flash ActiveX 控件

要将 Shockwave Flash ActiveX 控件包含在您自己的 Windows Forms 应用程序中，必须先将其添加到 Microsoft Visual Studio 工具箱中。

**将控件添加到工具箱中：**

1. 打开 Visual Studio 工具箱。
2. 右键单击 Visual Studio 2003 中的 Windows Forms 部分或 Visual Studio 2005 中的任何部分。在 Visual Studio 2003 中，从上下文菜单中选择 “添加 / 移除项”（Visual Studio 2005 中为 “选择项 ...”）。

   此操作将打开 “自定义工具箱” (2003)/ “选择工具箱项” (2005) 对话框。
3. 选择 “COM 组件”选项卡，该选项卡列出了计算机上所有可用的 COM 组件，包括 Flash Player ActiveX 控件。
4. 滚动到 Shockwave Flash Object 并选择它。

   若没有列出此项，请确保在系统中安装了 Flash Player ActiveX 控件。

## 了解 ActionScript 与 ActiveX 容器间的通信

使用外部 API 与 ActiveX 容器应用程序进行的通信类似于与 Web 浏览器之间的通信，但有一个重要的不同之处。如前所述，当 ActionScript 与 Web 浏览器通信时，对于开发人员而言，只需直接调用函数即可；有关如何设置函数调用和响应格式以在播放器和浏览器间进行传递的详细信息都会被隐藏。然而，使用外部 API 与 ActiveX 容器应用程序进行通信时，Flash Player 以特定的 XML 格式向应用程序发送消息 （函数调用和返回值），并要求来自容器应用程序的函数调用和返回值使用相同的 XML 格式。ActiveX 容器应用程序的开发人员编写的代码必须能够理解相应格式的函数调用和响应，以及应能够用相应格式创建它们。

Introvert IM C# 示例包含一组类，通过它们可以避免设置消息格式；相反，您可以在调用 ActionScript 函数和接收来自 ActionScript 的函数调用时，使用标准的数据类型。ExternalInterfaceProxy 类与其它辅助类一起提供该功能，并且可以在任何 .NET 项目中重复使用该类以便于进行外部 API 通信。

下面几段代码摘自主应用程序表单 (AppForm.cs)，说明使用 ExternalInterfaceProxy 类实现的更为简化的交互过程：

```
public class AppForm : System.Windows.Forms.Form
{
   ...
   private ExternalInterfaceProxy proxy;
   ...
   public AppForm()
   {
      ...
      // 注册此应用程序以便在代理收到
      // 来自 ActionScript 的调用时接收通知。
      proxy = new ExternalInterfaceProxy(IntrovertIMApp);
      proxy.ExternalInterfaceCall += new
   ExternalInterfaceCallEventHandler(proxy_ExternalInterfaceCall);
      ...
   }
   ...
```

该应用程序声明和创建名为 `proxy` 的 ExternalInterfaceProxy 实例，传入对用户界面 (`IntrovertIMApp`) 中的 Shockwave Flash ActiveX 控件的引用。然后，代码注册 `proxy_ExternalInterfaceCall()` 方法以接收代理的 `ExternalInterfaceCall` 事件。当函数调用来自 Flash Player 时， ExternalInterfaceProxy 类将调度该事件。C# 代码接收和响应来自 ActionScript 的函数调用的途径便是订阅该事件。

当函数调用来自 ActionScript 时，ExternalInterfaceProxy 实例 (proxy) 接收调用，转换它的 XML 格式，并通知作为代理的 ExternalInterfaceCall 事件的侦听器的对象。对于 AppForm 类，由 proxy_ExternalInterfaceCall() 方法处理该事件，如下所示：

```
/// <summary>
/// 在 SWF 发起 ActionScript ExternalInterface 调用时
/// 由代理进行调用
/// </summary>
private object proxy_ExternalInterfaceCall(object sender,
ExternalInterfaceCallEventArgs e)
{
    switch (e.FunctionCall.FunctionName)
    {
        case "isReady":
            return isReady();
        case "setSWFIsReady":
            setSWFIsReady();
            return null;
        case "newMessage":
            newMessage((string)e.FunctionCall.Arguments[0]);
            return null;
        case "statusChange":
            statusChange();
            return null;
        default:
            return null;
    }
}
...
```

该方法传递一个 ExternalInterfaceCallEventArgs 实例 （在本例中名为 e）。而该对象具有一个 FunctionCall 属性，该属性是 ExternalInterfaceCall 类的实例。

ExternalInterfaceCall 实例是具有两个属性的简单值对象。 FunctionName 属性包含 ActionScript ExternalInterface.Call() 语句中指定的函数名。如果在 ActionScript 中添加了任何参数，则这些参数都包含在 ExternalInterfaceCall 对象的 Arguments 属性中。在本例中，处理事件的方法只是一个 switch 语句，其作用类似于流量管理器。FunctionName 属性 (e.FunctionCall.FunctionName) 的值决定了将调用 AppForm 类的哪个方法。

先前代码清单中的 switch 语句的分支说明常见的方法调用情况。例如，任何方法都必须向 ActionScript 返回一个值 （例如， isReady() 方法调用）或返回 null （就像其它方法调用那样）。在 newMessage() 方法调用 （传递参数 e.FunctionCall.Arguments[0]，即 Arguments 数组的第一个元素）中，说明了访问从 ActionScript 传递的参数的方法。

使用 ExternalInterfaceProxy 类从 C# 调用 ActionScript 函数比从 ActionScript 接收函数调用更为简单明了。要调用 ActionScript 函数，请使用 ExternalInterfaceProxy 实例的 Call() 方法，如下所示：

```
    /// <summary>
    /// 在按下 “发送”按钮时进行调用；
    /// MessageText 文本字段中的值将作为参数传入。
    /// </summary>
    /// <param name="message"> 要发送的消息。</param>
    private void sendMessage(string message)
    {
        if (swfReady)
        {
            ...
            // 调用 ActionScript 中的 newMessage 函数。
            proxy.Call("newMessage", message);
        }
    }
    ...
    /// <summary>
    /// 调用 ActionScript 函数以获得当前的可用性状态
    /// 并将其写入文本字段。
    /// </summary>
    private void updateStatus()
    {
        Status.Text = (string)proxy.Call("getStatus");
    }
    ...
}
```

如本例所示，ExternalInterfaceProxy 类的 Call() 方法非常类似于 ActionScript 中的对应方法 ExternalInterface.Call()。第一个参数是一个字符串，即要调用的函数的名称。任何其它参数（未在此处显示）也一起传递给 ActionScript 函数。如果 ActionScript 函数返回值，则由 Call() 方法返回该值 （如前例所示）。

# 深入 ExternalInterfaceProxy 类内部

使用 ActiveX 控件的代理包装不一定总是切实可行的，或者您可能想自行编写代理类 （例如，使用不同的编程语言或针对不同的平台）。尽管有关创建代理的所有细节不会在此一一介绍，但是了解本例中代理类的内部运作原理却非常有益。

可以使用 Shockwave Flash ActiveX 控件的 CallFunction() 方法，通过外部 API 从 ActiveX 容器中调用 ActionScript 函数。以下从 ExternalInterfaceProxy 类的 Call() 方法摘录的代码显示了此过程：

```
// 对“_flashControl”中的 SWF 调用一个 ActionScript 函数，
//  “_flashControl”是一个 Shockwave Flash ActiveX 控件。
string response = _flashControl.CallFunction(request);
```

在摘录的这段代码中，`_flashControl` 为 Shockwave Flash ActiveX 控件。使用 `CallFunction()` 方法执行 ActionScript 函数调用。该方法具有一个参数（本例中为 `request`），该参数是一个包含 XML 格式指令（包括要调用的 ActionScript 函数名称及所有参数）的字符串。从 ActionScript 返回的任何值都被编码为 XML 格式字符串，并作为 `CallFunction()` 调用的返回值发送回来。在本例中， XML 字符串存储在 `response` 变量中。

从 ActionScript 接收函数调用是一个多步骤过程。来自 ActionScript 的函数调用会导致 Shockwave Flash ActiveX 控件调度它的 FlashCall 事件，因此要从 SWF 文件接收调用的类（如 ExternalInterfaceProxy 类）需要为该事件定义一个处理函数。在 ExternalInterfaceProxy 类中，该事件处理函数被命名为 `_flashControl_FlashCall()`，并进行了注册以便在类构造函数中侦听该事件，如下所示：

```
private AxShockwaveFlash _flashControl;

public ExternalInterfaceProxy(AxShockwaveFlash flashControl)
{
  _flashControl = flashControl;
  _flashControl.FlashCall += new
  _IShockwaveFlashEvents_FlashCallEventHandler(_flashControl_FlashCall);
}
...
private void _flashControl_FlashCall(object sender,
  _IShockwaveFlashEvents_FlashCallEvent e)
{
  // 使用事件对象的 request 属性（“e.request”）
  // 来执行某些操作。
  ...
  // 将值返回给 ActionScript；
  // 必须首先将返回值编码为一个 XML 格式的字符串。
  _flashControl.SetReturnValue(encodedResponse);
}
```

该事件对象 (e) 具有一个 `request` 属性 (`e.request`)，该属性是一个 XML 格式的字符串，它包含有关函数调用的信息，如函数名和参数。容器可以使用该信息来决定要执行哪些代码。在 ExternalInterfaceProxy 类中，请求从 XML 格式转换为一个 ExternalInterfaceCall 对象，该对象以更易于访问的形式提供相同的信息。ActiveX 控件的 `SetReturnValue()` 方法用于将函数结果返回给 ActionScript 调用方；必须再次以外部 API 所用的 XML 格式对结果参数进行编码。

ActionScript 与承载 Shockwave Flash ActiveX 控件的应用程序间的通信使用特定的 XML 格式对函数调用和值进行编码。在 Introvert IM C# 示例中，ExternalInterfaceProxy 类使应用程序表单中的代码能够直接对发送到 ActionScript 或从中接收的值执行运算，并忽略 Flash Player 所用的 XML 格式的细节。为了完成该操作，ExternalInterfaceProxy 类使用 ExternalInterfaceSerializer 类的方法来将 XML 消息实际转换为 .NET 对象。ExternalInterfaceSerializer 类具有四个公共方法：

- `EncodeInvoke()`：将函数名和 C# ArrayList 类型的参数编码为相应的 XML 格式。
- `EncodeResult()`：将结果值编码为相应的 XML 格式。
- `DecodeInvoke()`：对来自 ActionScript 的函数调用进行解码。 FlashCall 事件对象的 `request` 属性被传递给 `DecodeInvoke()` 方法，而且它将调用转换为一个 ExternalInterfaceCall 对象。
- `DecodeResult()`：对作为 ActionScript 函数调用的结果接收的 XML 进行解码。

这些方法将 C# 值编码为外部 API 的 XML 格式，并将 XML 解码为 C# 对象。有关 Flash Player 使用的 XML 格式的详细信息，请参阅第 634 页的 “外部 API 的 XML 格式”。

第 26 章

# Flash Player 安全性

26

安全性是 Adobe、用户、网站所有者和内容开发人员关注的焦点。因此， Adobe Flash Player 9 包含了一组安全性规则和控制，以保护用户、网站所有者和内容开发人员的利益。本章讨论了如何在开发 Flash 应用程序时使用 Flash Player 安全模型。在本章中，除非另有说明，否则假定所述的全部 SWF 文件均使用 ActionScript 3.0 发布（因此要在 Flash Player 9 或更高版本中运行）。

本章旨在对安全性进行一个总的说明，并不全面阐述所有的实现细节、用法方案或使用某些 API 所产生的分歧。有关 "Flash Player 安全性" 概念的更多详细论述，请参阅 "Flash Player 9 安全性白皮书"（位于 www.adobe.com/go/fp9_0_security_cn）。

## 目录

# Flash Player 安全性概述

Flash Player 安全性大部分基于加载的 SWF 文件、媒体和其它资源的原始域。来自特定 Internet 域（例如 www.example.com）的 SWF 文件始终可以访问该域的所有数据。这些资源放置在相同的安全分组中，该分组称为 “安全沙箱”。（有关详细信息，请参阅第 660 页的 “安全沙箱”。）

例如， SWF 文件可以加载 SWF 文件、位图、音频、文本文件以及自身域中的任何其它资源。此外，只要同一域中的两个 SWF 文件都是使用 ActionScript 3.0 编写的，则始终可以在这两个文件之间执行跨脚本访问操作。“跨脚本访问”是指一个 SWF 文件能够使用 ActionScript 访问另一个 SWF 文件中的属性、方法和对象。对于使用 ActionScript 3.0 编写的 SWF 文件与使用 ActionScript 早期版本编写的 SWF 文件，它们之间不支持跨脚本访问；但是，可以通过使用 LocalConnection 类在这些文件之间进行通信。有关详细信息，请参阅第 668 页的 “跨脚本访问”。

默认情况下，以下基本安全性规则始终适用：

- 位于相同安全沙箱中的资源始终可以互相访问。
- 远程沙箱中的 SWF 文件始终不能访问本地文件和数据。

Flash Player 将以下地址视为单个域，并为每个地址设置单独的安全沙箱：

- `http://example.com`
- `http://www.example.com`
- `http://store.example.com`
- `https://www.example.com`
- `http://192.0.34.166`

即使某个指定域 （例如 http://example.com）映射到特定 IP 地址 （例如 http://192.0.34.166）， Flash Player 也会为它们设置单独的安全沙箱。

开发人员可以使用两种基本方法为 SWF 文件授予访问权限，使之能够访问除该 SWF 文件所属沙箱之外的其它沙箱中的资源：

- `Security.allowDomain()` 方法 （请参阅第 659 页的 “作者 （开发人员）控制”）
- 跨域策略文件 （请参阅第 656 页的 “Web 站点控制 （跨域策略文件）”）

默认情况下，SWF 文件不能对其它域中的 ActionScript 3.0 SWF 文件执行跨脚本访问操作，也不能加载其它域中的数据。可以通过在加载的 SWF 文件中调用 `Security.allowDomain()` 方法来授予这种权限。有关详细信息，请参阅第 668 页的 “跨脚本访问”。

在 Flash Player 安全模型中，加载*内容*与访问或加载*数据*之间存在区别：

- 加载内容 — “内容”被定义为媒体，包括 Flash Player 可以播放的可视化媒体、音频、视频或包含显示媒体的 SWF 文件。“数据”被定义为只有 ActionScript 代码才能访问的内容。可以使用 Loader、 Sound 和 NetStream 等这样一些类来加载内容。
- 访问数据内容或加载数据 — 可以通过两种方式来访问数据：一种是从加载的媒体内容中提取数据，一种是从外部文件 （例如 XML 文件）中直接加载数据。可以通过使用 Bitmap 对象、 `BitmapData.draw()` 方法、 `Sound.id3` 属性或者 `SoundMixer.computeSpectrum()` 方法从加载的媒体中提取数据，使用诸如 URLStream、 URLLoader、 Socket 和 XMLSocket 等类来加载数据。

Flash Player 安全模型针对加载内容和访问数据定义了不同的规则。通常，加载内容的限制要比访问数据的限制少一些。

通常，可以从任意位置加载内容（SWF 文件、位图、MP3 文件和视频），但是如果内容来自执行加载的 SWF 文件所在的域之外的域，则会将内容划分到单独的安全沙箱中。

下面是加载内容的一些限制：

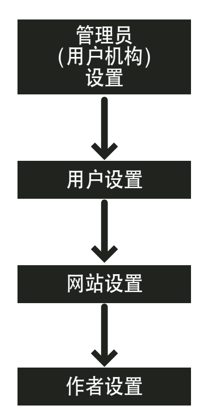
- 默认情况下，本地 SWF 文件 （从非网络地址加载的文件，例如用户硬盘上的文件）会被分类到只能与本地文件系统内容交互的沙箱中。这些文件无法从网络加载内容。有关详细信息，请参阅第 661 页的 “本地沙箱”。
- 实时消息传递协议 (RTMP) 服务器可以限制对内容的访问。有关详细信息，请参阅第 668 页的 “使用 RTMP 服务器传送的内容”。

如果加载的媒体为图像、音频或视频，则其安全沙箱之外的 SWF 文件无法访问该媒体的数据 （如像素数据和声音数据），除非该 SWF 文件的域已包含在该媒体原始域的跨域策略文件中。有关详细信息，请参阅第 671 页的 “作为数据访问加载的媒体”。

加载数据的其它格式包括文本文件或 XML 文件，这些文件可使用 URLLoader 对象来加载。同样，这种情况下要访问其它安全沙箱中的任何数据，必须通过原始域中的跨域策略文件来授予权限。有关详细信息，请参阅第 674 页的 “使用 URLLoader 和 URLStream”。

# 权限控制概述

Flash Player 客户端运行时安全模型是围绕 SWF 文件、本地数据和 Internet URL 等这些对象资源设计而成的模型。"资源持有者"是指拥有或使用这些资源的各方。资源持有者可以对其自己的资源进行控制（安全设置），每种资源有四个持有者。Flash Player 对这些控制严格采用一种权利层次结构，如下图所示：




*安全控制层次结构*

该图说明，如果管理员限制对资源的访问，则任何其他持有者都不能覆盖该限制。

以下各节详细说明了管理员、用户和网站控制。本章的其余部分对作者（开发人员）设置进行了说明。

# 管理用户控制

计算机的管理用户（使用管理权限登录的用户）可以应用能影响计算机所有用户的 Flash Player 安全设置。在非企业环境（例如家庭计算机）中，通常只有一个用户，该用户也拥有管理访问权限。即使是在企业环境中，单个用户也可以拥有计算机管理权限。

管理用户控制有两种类型：

- mms.cfg 文件
- "全局 Flash Player 信任"目录

## mms.cfg 文件

在 Mac OS X 系统上， mms.cfg 文件位于 /Library/Application Support/Macromedia 中。在 Microsoft Windows 系统上，该文件位于系统目录的 Macromedia Flash Player 文件夹中（例如，在 Windows XP 默认安装中为 C:\windows\system32\macromed\flash\mms.cfg）。

Flash Player 启动时将从此文件中读取其安全设置，然后使用这些设置限制功能。

mms.cfg 文件包括管理员用于执行以下任务的设置：

- **数据加载** — 限制读取本地 SWF 文件、禁止文件下载和上载以及对永久共享对象设置存储限制。
- **隐私控制** — 禁止麦克风和摄像头访问、禁止 SWF 文件播放无窗口内容，以及禁止与浏览器窗口中显示的 URL 不匹配的域中的 SWF 文件访问永久共享对象。
- **Flash Player 更新** — 设置检查 Flash Player 更新版本的时间间隔、指定检查 Flash Player 更新信息所使用的 URL、指定从其中下载 Flash Player 更新版本的 URL 以及完全禁用 Flash Player 的自动更新。
- **旧版文件支持** — 指定是否应将早期版本的 SWF 文件放置在受信任的本地沙箱中。
- **本地文件安全性** — 指定是否可以将本地文件放置在受信任的本地沙箱中。
- **全屏模式** — 禁用全屏模式。

SWF 文件可通过调用 `Capabilities.avHardwareDisable` 和 `Capabilities.localFileReadDisable` 属性来访问已禁用功能的某些信息。但是，mms.cfg 文件中的大部分设置无法通过 ActionScript 进行查询。

为对计算机强制执行与应用程序无关的安全和隐私策略，只能由系统管理员修改 mms.cfg 文件。 mms.cfg 文件不能用于安装应用程序。虽然使用管理权限运行的安装程序可以修改 mms.cfg 文件的内容，但是 Adobe 将此类使用视为违反用户的信任，并且劝告安装程序的创建者决不要修改 mms.cfg 文件。

## “全局 Flash Player 信任”目录

管理用户和安装应用程序可以将指定的本地 SWF 文件注册为受信任。这些 SWF 文件会被分配到受信任的本地沙箱。它们可以与任何其它 SWF 文件进行交互，也可以从任意位置（远程或本地）加载数据。文件在 “全局 Flash Player 信任”目录中被指定为受信任，该目录与 mms.cfg 文件的所在目录相同，位置（特定于当前用户）如下：

- Windows：system\Macromed\Flash\FlashPlayerTrust
  （例如， C:\windows\system32\Macromed\Flash\FlashPlayerTrust）
- Mac：app support/Macromedia/FlashPlayerTrust
  （例如， /Library/Application Support/Macromedia/FlashPlayerTrust）

"Flash Player 信任"目录可以包含任意数目的文本文件，每个文件均列出受信任的路径，一个路径占一行。每个路径可以是单个的 SWF 文件、 HTML 文件，也可以是目录。注释行以 # 号开头。例如，包含以下文本的 Flash Player 信任配置文件表示将向指定目录及所有子目录中的所有文件授予受信任状态：

```
# Trust files in the following directories:
C:\Documents and Settings\All Users\Documents\SampleApp
```

信任配置文件中列出的路径应始终是本地路径或 SMB 网络路径。信任配置文件中的任何 HTTP 路径均会被忽略；只能信任本地文件。

为避免发生冲突，应为每个信任配置文件指定一个与安装应用程序相应的文件名，并且使用 .cfg 文件扩展名。

由于开发人员通过安装应用程序分发本地运行的 SWF 文件，因此可以让安装应用程序向"全局 Flash Player 信任"目录添加一个配置文件，为要分发的文件授予完全访问权限。安装应用程序必须由拥有管理权限的用户来运行。与 mms.cfg 文件不同，包含 "全局 Flash Player 信任"目录是为了让安装应用程序授予信任权限。管理用户和安装应用程序都可以使用 "全局 Flash Player 信任"目录指定受信任的本地应用程序。

此外，还有适用于单个用户的 "Flash Player 信任"目录 （请参阅下一节 "用户控制"）。

## 用户控制

Flash Player 提供三种不同的用户级别权限设置机制："设置 UI"、"设置管理器"和 "用户 Flash Player 信任"目录。

### 设置 UI 和设置管理器

"设置 UI"是一种用于配置特定域设置的快速交互机制。"设置管理器"显示一个更详细的界面，并提供全局更改功能，全局更改可影响对许多域或所有域拥有的权限。另外，当 SWF 文件请求新的权限，要求有关安全或隐私的运行时决策时，程序会显示一些对话框，用户可以在这些对话框中调整某些 Flash Player 设置。

"设置管理器"和 "设置 UI"提供以下安全相关选项：

- 摄像头和麦克风设置 — 用户可以控制 Flash Player 对计算机上的摄像头和麦克风的访问。用户可以允许或拒绝对所有站点或特定站点的访问。如果用户没有为所有站点或特定站点指定设置，则当 SWF 文件试图访问摄像头或麦克风时，程序会显示一个对话框，让用户选择是否允许 SWF 文件访问该设备。用户也可以指定要使用的摄像头或麦克风，还可以设置麦克风的敏感度。

■ 共享对象存储设置 — 用户可以选择域能够用来存储永久共享对象的磁盘空间量。用户可以对任意数量的特定域进行这些设置，还可以为新域指定默认设置。默认限制是 100 KB 磁盘空间。有关永久共享对象的详细信息，请参阅《ActionScript 3.0 语言和组件参考》中的 SharedObject 类。

**注意** 在 mms.cfg 文件中所做的任何设置 （请参阅第 652 页的 “管理用户控制”）均不会反映在 “设置管理器”中。

有关 “设置管理器”的详细信息，请参阅 www.adobe.com/go/settingsmanager_cn。

## “用户 Flash Player 信任”目录

用户和安装应用程序可以将指定的本地 SWF 文件注册为受信任。这些 SWF 文件会被分配到受信任的本地沙箱。它们可以与任何其它 SWF 文件进行交互，也可以从任意位置（远程或本地）加载数据。用户在 “用户 Flash Player 信任”目录中将文件指定为受信任，该目录与 Flash 共享对象存储区域的所在目录相同，位置 （特定于当前用户）如下：

■ Windows：app data\Macromedia\Flash Player\#Security\FlashPlayerTrust
（例如， C:\Documents and Settings\JohnD\Application Data\Macromedia\Flash Player\#Security\FlashPlayerTrust）

■ Mac：app data/Macromedia/Flash Player/#Security/FlashPlayerTrust
（例如， /Users/JohnD/Library/Preferences/Macromedia/Flash Player/#Security/FlashPlayerTrust）

这些设置只会影响当前用户，不会影响登录到计算机的其他用户。如果没有管理权限的用户在属于他们自己的系统中安装了某个应用程序，则 “用户 Flash Player 信任”目录允许安装程序将该应用程序注册为该用户的受信任程序。

由于开发人员通过安装应用程序分发本地运行的 SWF 文件，因此可以让安装应用程序向 “用户 Flash Player 信任”目录添加一个配置文件，为要分发的文件授予完全访问权限。即使在这种情况下，也将 “用户 Flash Player 信任”目录文件视为用户控制，原因是用户操作（安装）启动了它。

此外，还有一个 “全局 Flash Player 信任”目录，管理用户或安装程序可使用该目录为所有计算机用户注册应用程序 （请参阅第 652 页的 “管理用户控制”）。

# Web 站点控制（跨域策略文件）

要使来自某个 Web 服务器的数据可用于来自其它域的 SWF 文件，可以在服务器上创建一个跨域策略文件。“跨域策略文件”是一个 XML 文件，它为服务器提供了一种方式，以指示该服务器的数据和文档可用于从某些域或所有域提供的 SWF 文件。服务器策略文件指定的域所提供的所有 SWF 文件都将被允许访问该服务器中的数据或资源。

跨域策略文件可影响对许多资源的访问，其中包括以下内容：

- 位图、声音和视频中的数据
- 加载 XML 和文本文件
- 对套接字和 XML 套接字连接的访问
- 将 SWF 文件从其它安全域导入到执行加载的 SWF 文件所在的安全域

本章的其余部分将对以上内容提供详细介绍。

## 策略文件语法

下面的示例显示了一个策略文件，该文件允许访问源自 `*.example.com`、`www.friendOfExample.com` 和 `192.0.34.166` 的 SWF 文件。

```
<?xml version="1.0"?>
<cross-domain-policy>
  <allow-access-from domain="*.example.com" />
  <allow-access-from domain="www.friendOfExample.com" />
  <allow-access-from domain="192.0.34.166" />
</cross-domain-policy>
```

当某个 SWF 文件试图访问另一个域中的数据时， Flash Player 会尝试自动从该域加载策略文件。如果试图访问数据的 SWF 文件所在的域包括在该策略文件中，则数据将自动成为可访问数据。

默认情况下，策略文件必须命名为 `crossdomain.xml`，并且必须位于服务器的根目录中。但是， SWF 文件可以通过调用 `Security.loadPolicyFile()` 方法检查是否为其它名称或位于其它目录中。跨域策略文件仅适用于从其中加载该文件的目录及其子目录。因此，根目录中的策略文件适用于整个服务器，但是从任意子目录加载的策略文件仅适用于该目录及其子目录。

策略文件仅影响对其所在特定服务器的访问。例如，位于 https://www.adobe.com:8080/crossdomain.xml 的策略文件只适用于在端口 8080 通过 HTTPS 对 www.adobe.com 进行的数据加载调用。

跨域策略文件包含单个 <cross-domain-policy> 标签，该标签又包含零个或多个 <allow-access-from> 标签。每个 <allow-access-from> 标签包含一个属性 domain，该属性指定一个确切的 IP 地址、一个确切的域或一个通配符域（任何域）。通配符域由单个星号 (*)（匹配所有域和所有 IP 地址）或后接后缀的星号（只匹配那些以指定后缀结尾的域）表示。后缀必须以点开头。但是，带有后缀的通配符域可以匹配那些只包含后缀但不包含前导点的域。例如， foo.com 会被看作是 *.foo.com 的一部分。 IP 域规范中不允许使用通配符。

如果您指定了一个 IP 地址，则只向使用 IP 语法从该 IP 地址加载的 SWF 文件（例如 http://65.57.83.12/flashmovie.swf）授予访问权限，而不向使用域名语法加载的 SWF 文件授予访问权限。 Flash Player 不执行 DNS 解析。

您可以允许访问来自任何域的文档，如下面的示例所示：

```
<?xml version="1.0"?>
<!-- http://www.foo.com/crossdomain.xml -->
<cross-domain-policy>
  <allow-access-from domain="*" />
</cross-domain-policy>
```

每个 <allow-access-from> 标签还具有可选的 secure 属性，其默认值为 true。如果您的策略文件在 HTTPS 服务器上，并且要允许非 HTTPS 服务器上的 SWF 文件从 HTTPS 服务器加载数据，则可以将此属性设置为 false。

将 secure 属性设置为 false 可能会危及 HTTPS 提供的安全性。特别是将此属性设置为 false 时，会使安全内容受到电子欺骗和窃听攻击。Adobe 强烈建议不要将 secure 属性设置为 false。

如果要加载的数据位于 HTTPS 服务器上，但是加载数据的 SWF 文件位于 HTTP 服务器上，则 Adobe 建议将要执行加载的 SWF 文件移动到 HTTPS 服务器上，以便可以使安全数据的所有副本都能得到 HTTPS 的保护。但是，如果决定必须将要执行加载的 SWF 文件保存在 HTTP 服务器上，则需将 secure="false" 属性添加到 <allow-access-from> 标签，如以下代码所示：

```
<allow-access-from domain="www.example.com" secure="false" />
```

不包含任何 <allow-access-from> 标签的策略文件相当于服务器上没有策略。

## 套接字策略文件

ActionScript 对象可实例化两种不同的服务器连接：基于文档的服务器连接和套接字连接。Loader、 Sound、 URLLoader 和 URLStream 等 ActionScript 对象可实例化基于文档的服务器连接，这些对象均根据 URL 加载文件。 ActionScript Socket 和 XMLSocket 对象进行套接字连接，这些对象操作的是数据流而非加载的文档。 Flash Player 支持两种策略文件：基于文档的策略文件和套接字策略文件。基于文档的连接需要基于文档的策略文件，套接字连接则需要套接字策略文件。

Flash Player 要求使用尝试连接希望使用的同类协议传输策略文件。例如，如果将策略文件放置在您的 HTTP 服务器上，则允许其它域中的 SWF 文件从该服务器（作为 HTTP 服务器）加载数据。但是，如果在这台服务器上未提供套接字策略文件，则禁止其它域的 SWF 文件在套接字级别连接到该服务器。检索套接字策略文件的方法必须与连接方法相匹配。

由套接字服务器提供的策略文件具有与任何其它策略文件相同的语法，只是前者还必须指定要对哪些端口授予访问权限。如果策略文件来自低于 1024 的端口号，则它可以对任何端口授予访问权限；如果策略文件来自 1024 或更高的端口，则它只能对 1024 端口和更高的端口授予访问权限。允许的端口在 `<allow-access-from>` 标签的 `to-ports` 属性中指定。单个端口号、端口范围和通配符都是允许值。

下面是一个 XMLSocket 策略文件示例：

```
<cross-domain-policy>
   <allow-access-from domain="*" to-ports="507" />
   <allow-access-from domain="*.example.com" to-ports="507,516" />
   <allow-access-from domain="*.example2.com" to-ports="516-523" />
   <allow-access-from domain="www.example2.com" to-ports="507,516-523" />
   <allow-access-from domain="www.example3.com" to-ports="*" />
</cross-domain-policy>
```

在 Flash Player 6 中首次引入策略文件时，并不支持套接字策略文件。与套接字服务器的连接由跨域策略文件所在默认位置中的一个策略文件授权，跨域策略文件位于与套接字服务器位于同一个域中的 HTTP 服务器的端口 80 上。为尽可能保留现有的服务器排列，Flash Player 9 仍然支持此功能。但是，Flash Player 现在的默认设置是在与套接字连接相同的端口上检索套接字策略文件。如果希望使用基于 HTTP 的策略文件来授权套接字连接，则必须使用如下所示代码显式请求 HTTP 策略文件：

```
Security.loadPolicyFile("http://socketServerHost.com/crossdomain.xml")
```

此外，为授权套接字连接，HTTP 策略文件只能来自跨域策略文件的默认位置，而非来自任何其它 HTTP 位置。从 HTTP 服务器获取的策略文件隐式向 1024 和所有更高端口授予套接字访问权限；HTTP 策略文件中的任何 `to-ports` 属性均被忽略。

有关套接字策略文件的详细信息，请参阅第 674 页的 “连接到套接字”。

## 预加载策略文件

从服务器加载数据或连接到套接字是一种异步操作，Flash Player 只是等待跨域策略文件完成下载，然后才开始主操作。但是，从图像中提取像素数据或从声音中提取采样数据是一种同步操作，跨域策略文件必须在可以提取数据之前先加载数据。加载媒体时，需要指定媒体检查是否存在跨域策略文件：

- 使用 `Loader.load()` 方法时，设置 `context` 参数的 `checkPolicyFile` 属性，该参数是一个 LoaderContext 对象。
- 使用 `<img>` 标签在文本字段中嵌入图像时，将 `<img>` 标签的 `checkPolicyFile` 属性设置为 `"true"`，如下所示：`<img checkPolicyFile = "true" src = "example.jpg">`。

- 使用 `Sound.load()` 方法时，设置 `context` 参数的 `checkPolicyFile` 属性，该参数是一个 SoundLoaderContext 对象。
- 使用 NetStream 类时，设置 NetStream 对象的 `checkPolicyFile` 属性。

设置上述参数时， Flash Player 首先会检查是否已经为该域下载了任何策略文件。然后考虑对 `Security.loadPolicyFile()` 方法的任何待定调用，以便查看它们是否在范围内，如果在范围内，则等待调用完成。然后，它查找服务器上默认位置中的跨域策略文件。

## 作者 （开发人员）控制

用于授予安全权限的主 ActionScript API 是 `Security.allowDomain()` 方法，它将向指定域中的 SWF 文件授予权限。在下面的示例中， SWF 文件向 www.example.com 域提供的 SWF 文件授予访问权限：

```
Security.allowDomain("www.example.com")
```

此方法为下列各项授予权限：

- SWF 文件之间的跨脚本访问 （请参阅第 668 页的 “跨脚本访问”）
- 显示列表访问 （请参阅第 670 页的 “遍历显示列表”）
- 事件检测 （请参阅第 671 页的 “事件安全性”）
- 对 Stage 对象的属性和方法的完全访问 （请参阅第 670 页的 “Stage 安全性”）

调用 `Security.allowDomain()` 方法的主要目的是为外部域中的 SWF 文件授予权限以访问调用 `Security.allowDomain()` 方法的 SWF 文件的脚本。有关详细信息，请参阅第 668 页的 “跨脚本访问”。

如果将 IP 地址指定为 `Security.allowDomain()` 方法的参数，则不允许任何源自该指定 IP 地址的访问方进行访问。相反，只允许 URL 中包含该指定 IP 地址的访问方进行访问，而不允许其域名映射到该 IP 地址的访问方进行访问。例如，如果域名 www.example.com 映射到 IP 地址 192.0.34.166，则对 `Security.allowDomain("192.0.34.166")` 的调用不会授予对 www.example.com 的访问权限。

可以将通配符 “*” 传递给 `Security.allowDomain()` 方法以允许从所有域进行访问。由于这种方式会为 “所有” 域中的 SWF 文件授予访问执行调用的 SWF 文件的脚本的权限，因此请谨慎使用通配符 “*”。

ActionScript 还包括一个权限 API，称为 `Security.allowInsecureDomain()`。此方法与 `Security.allowDomain()` 方法的作用相同，只是从安全 HTTPS 连接提供的 SWF 文件调用时，此方法还会允许非安全协议（例如 HTTP）提供的其它 SWF 文件访问执行调用的 SWF 文件。但是，在安全协议 (HTTPS) 中的文件与非安全协议（例如 HTTP）中的文件之间执行脚本访问操作并不是一种好的安全性做法；这样做会使安全内容受到电子欺骗和窃听攻击。下面是此类攻击的作用方式：由于 `Security.allowInsecureDomain()` 方法允许通过 HTTP 连接提供的 SWF 文件访问安全 HTTPS 数据，因此介入 HTTP 服务器和用户之间的攻击者能够将 HTTP SWF 文件替换为它们自己的文件，这样便可访问您的 HTTPS 数据。

另一种与安全性相关的重要方法是 `Security.loadPolicyFile()` 方法，该方法可让 Flash Player 在非标准位置检查是否存在跨域策略文件。有关详细信息，请参阅第 656 页的“Web 站点控制（跨域策略文件）”。

# 安全沙箱

客户端计算机可以从很多来源（如外部 Web 站点或本地文件系统）中获取单个 SWF 文件。当 SWF 文件及其它资源（例如共享对象、位图、声音、视频和数据文件）加载到 Flash Player 中时， Flash Player 会根据这些文件和资源的来源单独地将其分配到安全沙箱中。以下各节介绍了 Flash Player 强制执行的规则，这些规则控制着给定沙箱内的 SWF 文件可以访问的内容。

有关安全沙箱的详细信息，请参阅 “Flash Player 9 安全性白皮书”。

## 远程沙箱

Flash Player 将来自 Internet 的资源（包括 SWF 文件）分类到单独的沙箱中，这些沙箱与各自 Web 站点原始域相对应。默认情况下，对这些文件授予访问其自身所在服务器中任何资源的权限。通过显式的 Web 站点许可和作者许可（例如跨域策略文件和 `Security.allowDomain()` 方法），可以允许远程 SWF 文件访问其它域的其它数据。有关详细信息，请参阅第 656 页的 “Web 站点控制（跨域策略文件）”和第 659 页的 “作者（开发人员）控制”。

远程 SWF 文件无法加载任何本地文件或资源。

有关详细信息，请参阅 “Flash Player 9 安全性白皮书”。

## 本地沙箱

“本地文件”是指通过使用 `file:` 协议或统一命名约定 (UNC) 路径引用的任何文件。本地 SWF 文件放置在三个本地沙箱中的一个内：

- 只能与本地文件系统内容交互的沙箱 — 出于安全性考虑，Flash Player 在默认情况下将所有本地 SWF 文件和资源放置在只能与本地文件系统内容交互的沙箱中。通过此沙箱，SWF 文件可以读取本地文件（例如通过使用 URLLoader 类），但是它们无法以任何方式与网络进行通信。这样可向用户保证本地数据不会泄漏到网络或以其它方式不适当地共享。
- 只能与远程内容交互的沙箱 — 编译 SWF 文件时，可以指定该文件作为本地文件运行时拥有网络访问权限（请参阅第 662 页的 “设置本地 SWF 文件的沙箱类型”）。这些文件放置在只能与远程内容交互的沙箱中。分配到只能与远程内容交互的沙箱中的 SWF 文件将失去其本地文件访问权限，但允许这些 SWF 文件访问网络中的数据。不过，只有通过跨域策略文件或调用 `Security.allowDomain()` 方法来授予操作权限，才允许远程内容交互的 SWF 文件读取源自网络的数据。为授予此类权限，跨域策略文件必须向“所有”域授予权限，方法是使用 `<allow-access-from domain="*"/>` 或使用 `Security.allowDomain("*")`。有关详细信息，请参阅第 656 页的 “Web 站点控制（跨域策略文件）”和第 659 页的 “作者（开发人员）控制”。
- 受信任的本地沙箱 — 注册为受信任（由用户或安装程序注册）的本地 SWF 文件放置在受信任的本地沙箱中。系统管理员和用户还可以根据安全注意事项将本地 SWF 文件重新分配（移动）到受信任的本地沙箱，或者从该沙箱中进行重新分配（请参阅第 652 页的 “管理用户控制”和第 654 页的 “用户控制”）。分配到受信任的本地沙箱的 SWF 文件可以与其它任何 SWF 文件交互，也可以从任何位置（远程或本地）加载数据。

只能与远程内容交互的沙箱和只能与本地文件系统内容交互的沙箱之间的通信以及只能与本地文件系统内容交互的沙箱和远程沙箱之间的通信是严格禁止的。Flash 应用程序或用户/管理员不能授予允许此类通信的权限。

在本地 HTML 文件和本地 SWF 文件之间以任一方向访问脚本（例如使用 ExternalInterface 类）均要求涉及的 HTML 文件和 SWF 文件应位于受信任的本地沙箱中。这是因为浏览器的本地安全模型与 Flash Player 本地安全模型不同。

只能与远程内容交互的沙箱中的 SWF 文件无法加载只能与本地文件系统内容交互的沙箱中的 SWF 文件。只能与本地文件系统内容交互的沙箱中的 SWF 文件无法加载只能与远程内容交互的沙箱中的 SWF 文件。

## 设置本地 SWF 文件的沙箱类型

通过在 Adobe Flash CS3 创作工具中设置文档发布设置，您可以配置只能与本地文件系统内容交互的沙箱或只能与远程内容交互的沙箱的 SWF 文件。有关详细信息，请参阅《使用 Flash》中的 “为 Flash SWF 文件格式设置发布选项”。

最终用户或计算机管理员可以指定某个本地 SWF 文件是受信任的，以允许该文件从所有域（本地和网络）加载数据。这一点在 “全局 Flash Player 信任”目录和 “用户 Flash Player 信任”目录中指定。有关详细信息，请参阅第 652 页的“管理用户控制”和第 654 页的“用户控制”。

有关本地沙箱的详细信息，请参阅第 661 页的 “本地沙箱”。

## Security.sandboxType 属性

SWF 文件的作者可以使用只读的静态 `Security.sandboxType` 属性来确定 Flash Player 向其分配该 SWF 文件的沙箱类型。Security 类包括表示 `Security.sandboxType` 属性可能值的常量，如下所示：

- `Security.REMOTE` — SWF 文件来自 Internet URL，并遵守基于域的沙箱规则。
- `Security.LOCAL_WITH_FILE` — SWF 文件是本地文件，但尚未受到用户信任，且没有使用网络名称进行发布。此 SWF 文件可以从本地数据源读取数据，但无法与 Internet 进行通信。
- `Security.LOCAL_WITH_NETWORK` — SWF 文件是本地文件，且尚未受到用户信任，但已使用网络名称进行发布。此 SWF 文件可与 Internet 通信，但不能从本地数据源读取数据。
- `Security.LOCAL_TRUSTED` — SWF 文件是本地文件，且已使用 “设置管理器”或 Flash Player 信任配置文件受到用户信任。此 SWF 文件既可以从本地数据源读取数据，也可以与 Internet 进行通信。

# 限制网络 API

可以控制 SWF 文件对网络功能的访问，方法是通过在包含 SWF 内容的 HTML 页面的 `<object>` 和 `<embed>` 标签中设置 `allowNetworking` 参数。

`allowNetworking` 的可能值包括：

- “`all`”（默认值）— SWF 文件中允许使用所有网络 API。
- “`internal`” — SWF 文件可能不调用浏览器导航或浏览器交互 API （在本节后面部分中列出），但是它会调用任何其它网络 API。
- “`none`” — SWF 文件可能不调用浏览器导航或浏览器交互 API （在本节后面部分中列出），并且它无法使用任何 SWF 到 SWF 通信 API （也在本节后面部分中列出）。

调用被禁止的 API 会引发 SecurityError 异常。

要设置 allowNetworking 参数，在包含 SWF 文件引用的 HTML 页面的 <object> 和 <embed> 标签中添加 allowNetworking 参数并设置参数值，如下面的示例中所示：

```
<object classid="clsid:d27cdb6e-ae6d-11cf-96b8-444553540000"
 codebase="http://fpdownload.macromedia.com/pub/shockwave/cabs/flash/
  swflash.cab#version=9,0,18,0"
 width="600" height="400" id="test" align="middle">
<param name="allowNetworking" value="none" />
<param name="movie" value="test.swf" />
<param name="bgcolor" value="#333333" />
<embed src="test.swf" allowNetworking="none" bgcolor="#333333"
  width="600" height="400"
  name="test" align="middle" type="application/x-shockwave-flash"
  pluginspage="http://www.macromedia.com/go/getflashplayer_cn" />
</object>
```

HTML 页面也可能会使用脚本来生成 SWF 嵌入式标签。您需要更改该脚本，以便让它能够插入适当的 allowNetworking 设置。由 Flash 和 Adobe Flex Builder 生成的 HTML 页面使用 AC_FL_RunContent() 函数嵌入 SWF 文件的引用，您需要将 allowNetworking 参数设置添加到该脚本，如下所示：

```
AC_FL_RunContent( ... "allowNetworking", "none", ...)
```

当 allowNetworking 设置为 "internal" 时，以下 API 被禁止：

- navigateToURL()
- fscommand()
- ExternalInterface.call()

当 allowNetworking 设置为 "none" 时，除了上面列出的那些 API 外，还会禁止以下 API：

- sendToURL()
- FileReference.download()
- FileReference.upload()
- Loader.load()
- LocalConnection.connect()
- LocalConnection.send()
- NetConnection.connect()
- NetStream.play()
- Security.loadPolicyFile()
- SharedObject.getLocal()
- SharedObject.getRemote()
- Socket.connect()
- Sound.load()

- `URLLoader.load()`
- `URLStream.load()`
- `XMLSocket.connect()`

即使所选 `allowNetworking` 设置允许 SWF 文件使用网络 API，但根据安全沙箱的限制，还可能存在其它限制 （如本章所述）。

当 `allowNetworking` 设置为 “none” 时，无法在 TextField 对象 `htmlText` 属性的 `<img>` 标签中引用外部媒体 （这会引发 SecurityError 异常）。

当 `allowNetworking` 设置为 "none" 时，从导入的共享库添加到 Flash 创作工具 （而不是 ActionScript）中的元件在运行时被阻止。

# 全屏模式安全性

Flash Player 9.0.27.0 和更高版本支持全屏模式，在该模式中 Flash 内容可以填满整个屏幕。要进入全屏模式，需将 Stage 的 `displayState` 属性设置为 `StageDisplayState.FULL_SCREEN` 常量。有关详细信息，请参阅第 336 页的 “处理全屏模式”。

对于在浏览器中运行的 SWF 文件，存在一些安全注意事项。

要启用全屏模式，请在包含 SWF 文件引用的 HTML 页面的 `<object>` 和 `<embed>` 标签中添加 `allowFullScreen` 参数，并将参数值设置为 “true” （默认值为 “false” ），如下例所示：

```
<object classid="clsid:d27cdb6e-ae6d-11cf-96b8-444553540000"
 codebase="http://fpdownload.macromedia.com/pub/shockwave/cabs/flash/
  swflash.cab#version=9,0,18,0"
 width="600" height="400" id="test" align="middle">
<param name="allowFullScreen" value="true" />
<param name="movie" value="test.swf" />
<param name="bgcolor" value="#333333" />
<embed src="test.swf" allowFullScreen="true" bgcolor="#333333"
  width="600" height="400"
  name="test" align="middle" type="application/x-shockwave-flash"
  pluginspage="http://www.macromedia.com/go/getflashplayer_cn" />
</object>
```

HTML 页面也可能会使用脚本来生成 SWF 嵌入式标签。您必须更改该脚本，以便让它能够插入适当的 `allowFullScreen` 设置。由 Flash 和 Flex Builder 生成的 HTML 页面使用 `AC_FL_RunContent()` 函数嵌入 SWF 文件的引用，您需要添加 `allowFullScreen` 参数设置，如下所示：

```
AC_FL_RunContent( ... "allowFullScreen", "true", ...)
```

仅当在响应鼠标事件或键盘事件时才会调用启动全屏模式的 ActionScript。如果在其它情况中调用， Flash Player 会引发异常。

在全屏模式下，用户无法在文本输入字段中输入文本。所有键盘输入和键盘相关的 ActionScript 在全屏模式下均会被禁用，但将应用程序返回标准模式的键盘快捷键 （例如按 Esc）除外。

当内容进入全屏模式时，程序会显示一条消息，指导用户如何退出和返回标准模式。该消息将显示几秒钟，然后淡出。

如果某个调用方与 Stage 所有者（主 SWF 文件）没有位于同一安全沙箱，则调用 Stage 对象的 `displayState` 属性会引发异常。有关详细信息，请参阅第 670 页的"Stage 安全性"。

管理员可以通过在 mms.cfg 文件中设置 `FullScreenDisable = 1` 对浏览器中运行的 SWF 文件禁用全屏模式。有关详细信息，请参阅第 652 页的 "管理用户控制"。

在浏览器中，必须在 HTML 页面中包含 SWF 文件，才能进入全屏模式。

在独立的播放器或放映文件中始终允许全屏模式。

# 加载内容

SWF 文件可以加载以下内容类型：

- SWF 文件
- 图像
- 声音
- 视频

## 加载 SWF 文件和图像

使用 Loader 类加载 SWF 文件和图像（JPG、GIF 或 PNG 文件）。除只能与本地文件系统内容交互的沙箱中的 SWF 文件之外，其它所有 SWF 文件都可以从任何网络域加载 SWF 文件和图像。只有本地沙箱中的 SWF 文件才能从本地文件系统中加载 SWF 文件和图像。但是，只能与远程内容交互的沙箱中的文件只能加载位于受信任的本地沙箱或只能与远程内容交互的沙箱中的本地 SWF 文件。只能与远程内容交互的沙箱中的 SWF 文件可加载非 SWF 文件 （例如图像）的本地内容，但是无法访问所加载内容中的数据。

从不受信任的来源 （如 Loader 对象的根 SWF 文件所在域以外的域）加载 SWF 文件时，您可能需要为 Loader 对象定义遮罩，以防止加载的内容 （Loader 对象的子级）绘制到该遮罩之外的 Stage 部分中，如以下代码所示：

```
import flash.display.*;
import flash.net.URLRequest;
var rect:Shape = new Shape();
rect.graphics.beginFill(0xFFFFFF);
rect.graphics.drawRect(0, 0, 100, 100);
addChild(rect);
var ldr:Loader = new Loader();
```

```
ldr.mask = rect;
var url:String = "http://www.unknown.example.com/content.swf";
var urlReq:URLRequest = new URLRequest(url);
ldr.load(urlReq);
addChild(ldr);
```

当调用 Loader 对象的 `load()` 方法时，可以指定一个 `context` 参数，该参数是一个 LoaderContext 对象。 LoaderContext 类包括三个属性，用于定义如何使用加载的内容的上下文：

- `checkPolicyFile`：仅当加载图像文件 （不是 SWF 文件）时才会使用此属性。如果图像文件所在的域与包含 Loader 对象的文件所在的域不同，则指定此属性。如果将此属性设置为 `true`， Loader 将检查跨域策略文件的原始服务器 （请参阅第 656 页的 "Web 站点控制 （跨域策略文件）"）。如果服务器授予 Loader 域适当权限，则来自 Loader 域中 SWF 文件的 ActionScript 可以访问所加载图像中的数据。换言之，可以使用 `Loader.content` 属性获取对表示所加载图像的 Bitmap 对象的引用，或使用 `BitmapData.draw()` 方法访问所加载图像中的像素。
- `securityDomain`：仅当加载 SWF 文件 （不是图像）时才会使用此属性。如果 SWF 文件所在的域与包含 Loader 对象的文件所在的域不同，则指定此属性。对于 `securityDomain` 属性而言，目前仅支持以下两个值：`null` （默认值）和 `SecurityDomain.currentDomain`。如果指定 `SecurityDomain.currentDomain`，则要求加载的 SWF 文件应 "导入"到执行加载的 SWF 文件所在的沙箱中，这意味着其运行方式就像它已从执行加载的 SWF 文件自己的服务器中加载一样。只有在位于加载的 SWF 文件服务器上找到跨域策略文件时才允许这样做，从而允许执行加载的 SWF 文件所在的域进行访问。如果找到所需的策略文件，则一旦加载开始，加载方和被加载方可以自由地互相访问脚本，原因是它们位于同一沙箱中。请注意，多数情况可以通过执行普通加载操作然后让加载的 SWF 文件调用 `Security.allowDomain()` 方法来取代沙箱导入。由于加载的 SWF 文件将位于自己的原始沙箱中，并因而能够访问自己实际服务器上的资源，因此后一种方法会更易于使用。
- `applicationDomain`：仅当加载使用 ActionScript 3.0 编写的 SWF 文件 （不是图像或使用 ActionScript 1.0 或 2.0 编写的 SWF 文件）时才会使用此属性。当加载文件时，可以指定文件应放置在特定的应用程序域中，而不是默认放置在一个新的应用程序域中，这个新的应用程序域是执行加载的 SWF 文件所在应用程序域的子域。请注意，应用程序域是安全域的子单位，因此仅当要加载的 SWF 文件由于以下原因来自您自己的安全域时，才能指定目标应用程序域：该文件来自您自己的服务器，或者使用 `securityDomain` 属性已成功地将该文件导入到您的安全域中。如果指定应用程序域，但加载的 SWF 文件属于其它安全域，则在 `applicationDomain` 中指定的域将被忽略。有关详细信息，请参阅第 603 页的 "使用 ApplicationDomain 类"。

有关详细信息，请参阅第 362 页的 "指定加载上下文"。

Loader 对象的一个重要属性就是 contentLoaderInfo 属性，该属性是一个 LoaderInfo 对象。与大部分对象不同， LoaderInfo 对象在执行加载的 SWF 文件和被加载的内容之间共享，并且双方始终可以访问该对象。当被加载的内容为 SWF 文件时，它可以通过 DisplayObject.loaderInfo 属性访问 LoaderInfo 对象。 LoaderInfo 对象包括诸如加载进度、加载方和被加载方的 URL、加载方和被加载方之间的信任关系等信息及其它信息。有关详细信息，请参阅第 361 页的 “监视加载进度”。

## 加载声音和视频

除只能与本地文件系统内容交互的沙箱中的那些 SWF 文件之外，所有 SWF 文件都允许从网络来源加载声音和视频，使用 Sound.load()、 NetConnection.connect() 和 NetStream.play() 方法即可。

只有本地 SWF 文件才能从本地文件系统加载媒体。只有只能与本地文件系统内容交互的沙箱或受信任的本地沙箱中的 SWF 文件才能访问这些加载文件中的数据。

对加载的媒体还存在一些其它数据访问限制。有关详细信息，请参阅第 671 页的 “作为数据访问加载的媒体”。

## 使用文本字段中的 <img> 标签加载 SWF 文件和图像

通过使用 <img> 标签，可以将 SWF 文件和位图加载到文本字段中，如以下代码所示：

```
<img src = 'filename.jpg' id = 'instanceName' >
```

通过使用 TextField 实例的 getImageReference() 方法，可以访问以这种方式加载的内容，如以下代码所示：

```
var loadedObject:DisplayObject = 
  myTextField.getImageReference('instanceName');
```

但是请注意，以这种方式加载的 SWF 文件和图像会被放入与各自来源相应的沙箱中。

当在文本字段中使用 <img> 标签加载图像文件时，通过跨域策略文件可以允许访问图像中的数据。通过将 checkPolicyFile 属性添加到 <img> 标签上，可以检查是否存在策略文件，如以下代码所示：

```
<img src = 'filename.jpg' checkPolicyFile = 'true' id = 'instanceName' >
```

当在文本字段中使用 <img> 标签加载 SWF 时，可以允许通过调用 Security.allowDomain() 方法来访问该 SWF 文件的数据。

当在文本字段中使用 <img> 标签加载外部文件时（相对于使用嵌在 SWF 文件中的 Bitmap 类），会自动创建一个 Loader 对象作为 TextField 对象的子对象，并且会将外部文件加载到该 Loader 对象中，就如同使用了 ActionScript 中的 Loader 对象来加载文件一样。在这种情况下， getImageReference() 方法返回自动创建的 Loader。由于此 Loader 对象与调用代码位于同一安全沙箱中，因此访问此对象不需要任何安全检查。

但是，当引用 Loader 对象的 `content` 属性来访问加载的媒体时，需要应用安全性规则。如果内容是图像，则需要实现跨域策略文件；如果内容是 SWF 文件，则需要让 SWF 文件中的代码调用 `allowDomain()` 方法。

## 使用 RTMP 服务器传送的内容

Flash Media Server 使用实时媒体协议 (RTMP) 提供数据、音频和视频。SWF 文件通过使用 NetConnection 类的 `connect()` 方法并作为参数传递 RTMP URL 来加载此媒体。Flash Media Server 可以根据所请求文件的域来限制连接并防止内容被下载。有关详细信息，请参阅 Flash Media Server 文档。

对于从 RTMP 源加载的媒体，不能使用 `BitmapData.draw()` 和 `SoundMixer.computeSpectrum()` 方法来提取运行时图形和声音数据。

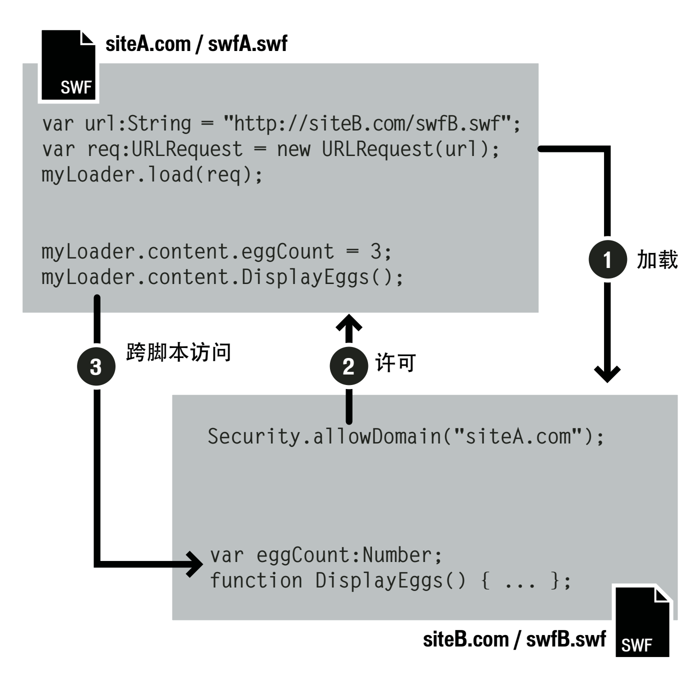
# 跨脚本访问

如果两个使用 ActionScript 3.0 编写的 SWF 文件来自同一个域，例如，一个 SWF 文件的 URL 是 http://www.example.com/swfA.swf，另一个文件的 URL 是 http://www.example.com/swfB.swf，则一个 SWF 文件可以检查并修改另一个 SWF 文件中的变量、对象、属性、方法等等，反之亦然。这称为 "跨脚本访问"。

在 AVM1 SWF 文件和 AVM2 SWF 文件之间不支持跨脚本访问。AVM1 SWF 文件是使用 ActionScript 1.0 或 ActionScript 2.0 创建的文件。（AVM1 和 AVM2 指的是 ActionScript 虚拟机。）但是，可以使用 LocalConnection 类在 AVM1 和 AVM2 之间发送数据。

如果两个使用 ActionScript 3.0 编写的 SWF 文件来自不同的域 （例如，http://siteA.com/swfA.swf 和 http://siteB.com/swfB.swf），则在默认情况下， Flash Player 既不允许 swfA.swf 访问 swfB.swf 的脚本，也不允许 swfB.swf 访问 swfA.swf 的脚本。通过调用 `Security.allowDomain()`，一个 SWF 文件可向其它域中的 SWF 文件授予访问其脚本的权限。通过调用 `Security.allowDomain("siteA.com")`， swfB.swf 向来自 siteA.com 的 SWF 文件授予访问其脚本的权限。

在任何跨域的情况下，明确所涉及的双方非常重要。为了便于进行此讨论，我们将执行跨脚本访问的一方称为“访问方”（通常是执行访问的 SWF），将另一方称为“被访问方”（通常是被访问的 SWF）。当 siteA.swf 访问 siteB.swf 的脚本时， siteA.swf 是访问方，siteB.swf 是被访问方，如下图所示：




使用 `Security.allowDomain()` 方法建立的跨域权限是不对称的。在上例中， siteA.swf 可以访问 siteB.swf 的脚本，但 siteB.swf 无法访问 siteA.swf 的脚本，这是因为 siteA.swf 未调用 `Security.allowDomain()` 方法来授予 siteB.com 中的 SWF 文件访问其脚本的权限。可以通过使两个 SWF 文件都调用 `Security.allowDomain()` 方法来设置对称权限。

除了防止 SWF 文件受到其它 SWF 文件发起的跨域脚本访问外，Flash Player 还防止 SWF 文件受到 HTML 文件发起的跨域脚本访问。可以通过 `ExternalInterface.addCallback()` 方法建立的回调执行 HTML 到 SWF 的脚本访问。当 HTML 到 SWF 的脚本访问跨域时，被访问的 SWF 文件必须调用 `Security.allowDomain()` 方法（这与访问方是 SWF 文件时一样），否则操作将失败。有关详细信息，请参阅第 659 页的 “作者 （开发人员）控制”。

此外， Flash Player 还对 SWF 到 HTML 的脚本访问提供安全控制。有关详细信息，请参阅第 678 页的 “控制对主机网页中脚本的访问”。

# Stage 安全性

Stage 对象的某些属性和方法可用于显示列表中的任何 sprite 或影片剪辑。

但是，我们说 Stage 对象有一个所有者，即加载的第一个 SWF 文件。默认情况下， Stage 对象的以下属性和方法只能用于与舞台所有者位于同一安全沙箱中的 SWF 文件：

| 属性 | | 方法 |
|---|---|---|
| align | showDefaultContextMenu | addChild() |
| displayState | stageFocusRect | addChildAt() |
| frameRate | stageHeight | addEventListener() |
| height | stageWidth | dispatchEvent() |
| mouseChildren | tabChildren | hasEventListener() |
| numChildren | textSnapshot | setChildIndex() |
| quality | width | willTrigger() |
| scaleMode | | |

为使与 Stage 所有者不在同一沙箱中的 SWF 文件能够访问这些属性和方法，舞台所有者 SWF 文件必须调用 Security.allowDomain() 方法来允许外部沙箱的域。有关详细信息，请参阅第 659 页的 “作者 （开发人员）控制”。

frameRate 属性是一种特殊情况：任何 SWF 文件均能读取 frameRate 属性。但是，只有位于 Stage 所有者的安全沙箱中的文件（或通过调用 Security.allowDomain() 方法被授予权限的文件）才能更改该属性。

此外，对于 Stage 对象的 removeChildAt() 和 swapChildrenAt() 方法还存在一些限制，但是这些限制与其它限制不同。要调用这些方法，不需要与 Stage 所有者位于同一域中，而是代码必须与受影响的子对象位于同一域中，或者子对象可以调用 Security.allowDomain() 方法。

# 遍历显示列表

一个 SWF 文件能够访问从其它沙箱中加载的显示对象受到一定限制。为使 SWF 文件能够访问由其它沙箱中的另一个 SWF 文件创建的显示对象，被访问的 SWF 文件必须调用 Security.allowDomain() 方法向进行访问的 SWF 文件所在的域授予访问权限。有关详细信息，请参阅第 659 页的 “作者 （开发人员）控制”。

要访问由 Loader 对象加载的 Bitmap 对象，图像文件的原始服务器上必须存在跨域策略文件，并且该跨域策略文件必须为尝试访问该 Bitmap 对象的 SWF 文件所在的域授予访问权限 （请参阅第 656 页的 “Web 站点控制 （跨域策略文件）”）。

与加载的文件（和 Loader 对象）相对应的 LoaderInfo 对象包括以下三种属性（定义加载的对象和 Loader 对象之间的关系）：`childAllowsParent`、`parentAllowsChild` 和 `sameDomain`。

## 事件安全性

根据调度事件的显示对象所在的沙箱，与显示列表相关的事件具有一定的安全性访问限制。显示列表中的事件具有冒泡阶段和捕获阶段（在第 267 页的第 10 章“处理事件”中论述）。在冒泡阶段和捕获阶段期间，事件从源显示对象开始迁移，并经过显示列表中的父显示对象。如果父对象与源显示对象位于不同的安全沙箱中，则捕获阶段和冒泡阶段在父对象的下方停止，除非父对象的所有者与源对象的所有者互相信任。这种互相信任可以通过以下方式来建立：

1. 拥有父对象的 SWF 文件必须调用 `Security.allowDomain()` 方法，以信任拥有源对象的 SWF 文件所在的域。
2. 拥有源对象的 SWF 文件必须调用 `Security.allowDomain()` 方法，以信任拥有父对象的 SWF 文件所在的域。

与加载的文件（和 Loader 对象）相对应的 LoaderInfo 对象包括以下两种属性（定义加载的对象和 Loader 对象之间的关系）：`childAllowsParent` 和 `parentAllowsChild`。

对于从显示对象以外的对象调度的事件，不存在任何安全检查或与安全相关的含义。

# 作为数据访问加载的媒体

访问加载的数据可使用诸如 `BitmapData.draw()` 和 `SoundMixer.computeSpectrum()` 等方法。默认情况下，一个安全沙箱中的 SWF 文件无法从另一个沙箱中加载的媒体所呈现或播放的图形或音频对象中获取像素数据或音频数据。但是，可以使用以下方法授予这种权限：

- 在加载的 SWF 文件中，调用 `Security.allowDomain()` 方法可授予对其它域中 SWF 文件的数据访问权限。
- 对于加载的图像、声音或视频，在被加载文件所在的服务器上添加跨域策略文件。此策略文件必须向试图调用 `BitmapData.draw()` 或 `SoundMixer.computeSpectrum()` 方法的 SWF 文件所在的域授予访问权限，以便从文件提取数据。

以下各节提供有关访问位图、声音和视频数据的详细信息。

## 访问位图数据

借助 BitmapData 对象的 `draw()` 方法，可以将任何显示对象的当前显示像素绘制到 BitmapData 对象。这样能够包括 MovieClip 对象、Bitmap 对象或任何显示对象的像素。要使用 `draw()` 方法将像素绘制到 BitmapData 对象，必须满足以下条件：

- 如果是除所加载位图之外的源对象，则该源对象及其 （如果是 Sprite 或 MovieClip 对象）所有子对象必须与调用 `draw()` 方法的对象位于同一域，或者它们必须位于调用方通过调用 `Security.allowDomain()` 方法可访问的 SWF 文件中。
- 如果是已加载的位图源对象，则该源对象必须与调用 `draw()` 方法的对象位于同一域，或者其源服务器必须包含一个授予调用域访问权限的跨域策略文件。

如果不满足上述条件，则会引发 SecurityError 异常。

当使用 Loader 类的 `load()` 方法加载图像时，可以指定一个 `context` 参数，该参数是一个 LoaderContext 对象。如果将 LoaderContext 对象的 `checkPolicyFile` 属性设置为 `true`，则 Flash Player 将在从其中加载该图像的服务器上检查是否存在跨域策略文件。如果存在跨域策略文件且该文件允许执行加载的 SWF 文件所在的域进行访问，则会允许该文件访问 Bitmap 对象中的数据，否则就不允许。

此外，还可以通过文本字段中的 `<img>` 标签在加载的图像中指定 `checkPolicyFile` 属性。有关详细信息，请参阅第 667 页的 “使用文本字段中的 <img> 标签加载 SWF 文件和图像”。

## 访问声音数据

以下与声音相关的 ActionScript 3.0 API 存在一些安全限制：

- `SoundMixer.computeSpectrum()` 方法 — 对于与声音文件位于同一安全沙箱的 SWF 文件，始终允许使用该方法。对于其它沙箱中的文件，则需经过安全检查。
- `SoundMixer.stopAll()` 方法 — 对于与声音文件位于同一安全沙箱的 SWF 文件，始终允许使用该方法。对于其它沙箱中的文件，则需经过安全检查。
- Sound 类的 `id3` 属性 — 对于与声音文件位于同一安全沙箱的 SWF 文件，始终允许使用该属性。对于其它沙箱中的文件，则需经过安全检查。

每个声音都具有两种与之关联的沙箱 （一个内容沙箱和一个所有者沙箱）：

- 声音的源域确定内容沙箱，内容沙箱则确定是否可以通过声音的 `id3` 属性和 `SoundMixer.computeSpectrum()` 方法提取声音中的数据。
- 启动声音播放的对象确定所有者沙箱，所有者沙箱则确定是否可以使用 `SoundMixer.stopAll()` 方法停止声音。

当使用 Sound 类的 `load()` 方法加载声音时，可以指定一个 `context` 参数，该参数是一个 SoundLoaderContext 对象。如果将 SoundLoaderContext 对象的 `checkPolicyFile` 属性设置为 `true`，则 Flash Player 将在从其中加载该声音的服务器上检查是否存在跨域策略文件。如果存在跨域策略文件且该文件允许执行加载的 SWF 文件所在的域进行访问，则会允许该文件访问 Sound 对象的 `id` 属性，否则就不允许。此外，设置 `checkPolicyFile` 属性可以为加载的声音启用 `SoundMixer.computeSpectrum()` 方法。

可以使用 `SoundMixer.areSoundsInaccessible()` 方法确定对 `SoundMixer.stopAll()` 方法的调用是否会因调用方无法访问一个或多个声音所有者的沙箱而停止全部声音。

调用 `SoundMixer.stopAll()` 方法会停止与 `stopAll()` 的调用方位于同一所有者沙箱中的那些声音。它还会停止由调用 `Security.allowDomain()` 方法的 SWF 文件来启动播放的声音，以允许调用 `stopAll()` 方法的 SWF 文件所在的域进行访问。任何其它声音均不会停止，可以通过调用 `SoundMixer.areSoundsInaccessible()` 方法来确定此类声音是否存在。

调用 `computeSpectrum()` 方法要求播放的每个声音应与调用该方法的对象位于同一沙箱中，或者位于已向调用方的沙箱授予权限的源；否则会引发 SecurityError 异常。对于从 SWF 文件库中嵌入声音加载的声音，通过在加载的 SWF 文件中调用 `Security.allowDomain()` 方法授予权限。对于从非 SWF 文件的源（源自加载的 MP3 文件或 Flash 视频）中加载的声音，源服务器上的跨域策略文件将授予访问所加载媒体中数据的权限。如果声音是从 RTMP 数据流加载的，则无法使用 `computeSpectrum()` 方法。

有关详细信息，请参阅第 659 页的 “作者（开发人员）控制”和第 656 页的 “Web 站点控制（跨域策略文件）”。

## 访问视频数据

可以使用 `BitmapData.draw()` 方法捕获视频当前帧中的像素数据。

视频有两种不同形式：

- RTMP 视频
- 渐进式视频（该视频是从没有 RTMP 服务器的 FLV 文件加载的）

无法使用 `BitmapData.draw()` 方法访问 RTMP 视频。

当调用 `BitmapData.draw()` 方法，并且 `source` 参数为渐进式视频时，`BitmapData.draw()` 的调用方必须与 FLV 文件位于同一沙箱，或者 FLV 文件所在的服务器上必须存在一个策略文件，用以向执行调用的 SWF 文件所在的域授予访问权限。通过将 NetStream 对象的 `checkPolicyFile` 属性设置为 `true`，可以请求下载该策略文件。

# 加载数据

SWF 文件可以将服务器的数据加载到 ActionScript 中，也可以将 ActionScript 中的数据发送到服务器。加载数据与加载媒体的操作方式不同，原因是加载的信息将直接显示在 ActionScript 中，而不是作为媒体显示。通常，SWF 文件可以从它们自己的域中加载数据。但是，要从其它域加载数据，它们通常需要跨域策略文件。

## 使用 URLLoader 和 URLStream

可以加载诸如 XML 文件或文本文件等数据。URLLoader 和 URLStream 类的 `load()` 方法受到跨域策略文件权限的控制。

如果使用 `load()` 方法从与调用该方法的 SWF 文件所在域不同的域中加载内容，则 Flash Player 会在被加载资源所在的服务器上检查是否存在跨域策略文件。如果存在跨域策略文件，并且该文件向执行加载的 SWF 文件所在的域授予访问权限，则可以加载数据。

## 连接到套接字

默认情况下，禁用对套接字和 XML 套接字连接的跨域访问。此外，默认情况下还禁止访问与低于 1024 的端口上的 SWF 文件位于同一个域的套接字连接，但可以通过提供以下任一位置中的跨域策略文件来允许访问这些端口：

- 与主套接字连接相同的端口
- 不同端口
- 与套接字服务器位于同一个域的 HTTP 服务器的端口 80 上

如果提供的跨域策略文件与主套接字连接位于同一端口，或者位于不同端口，则通过在跨域策略文件中使用 `to-ports` 属性来枚举允许的端口，如下例所示：

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy
  SYSTEM "http://www.adobe.com/xml/dtds/cross-domain-policy.dtd">
<!-- Policy file for xmlsocket://socks.mysite.com -->
<cross-domain-policy>
   <allow-access-from domain="*" to-ports="507" />
   <allow-access-from domain="*.example.com" to-ports="507,516" />
   <allow-access-from domain="*.example.org" to-ports="516-523" />
   <allow-access-from domain="adobe.com" to-ports="507,516-523" />
   <allow-access-from domain="192.0.34.166" to-ports="*" />
</cross-domain-policy>
```

要检索与主套接字连接位于相同端口中的套接字策略文件，只需调用 `Socket.connect()` 或 `XMLSocket.connect()` 方法；并且，如果指定的域与执行调用的 SWF 文件所在的域不同，Flash Player 将自动尝试从正在尝试的主连接所在的相同端口中检索策略文件。要从与主连接位于同一服务器上的不同端口检索套接字策略文件，需使用特殊的 “xmlsocket” 语法调用 `Security.loadPolicyFile()` 方法，如下所示：

```
Security.loadPolicyFile("xmlsocket://server.com:2525");
```

先调用 `Security.loadPolicyFile()` 方法，然后再调用 `Socket.connect()` 或 `XMLSocket.connect()` 方法。 Flash Player 随后将一直等待完成策略文件请求，之后再决定是否允许主连接。

如果要实现套接字服务器，并且需要提供套接字策略文件，则应决定是使用接受主连接的同一端口提供策略文件，还是使用不同的端口来提供策略文件。无论是哪种情况，服务器均必须等待客户端的第一次传输之后再决定是发送策略文件还是建立主连接。当 Flash Player 请求策略文件时，它始终会在建立连接后传输以下字符串：

```
<policy-file-request/>
```

服务器收到此字符串后，即会传输该策略文件。程序对于策略文件请求和主连接并不会使用同一连接，因此应在传输策略文件后关闭连接。如果不关闭连接， Flash Player 将关闭策略文件连接，之后重新连接以建立主连接。

有关详细信息，请参阅第 657 页的 “套接字策略文件”。

# 发送数据

当 SWF 文件中的 ActionScript 代码向服务器或资源发送数据时，将会发生数据发送操作。对于网络域 SWF 文件，始终允许发送数据。本地 SWF 文件则只有在位于受信任的本地沙箱或只能与远程内容交互的沙箱中时，才能向网络地址发送数据。有关详细信息，请参阅第 661 页的 “本地沙箱”。

可以使用 `flash.net.sendToURL()` 函数向 URL 发送数据。还可以使用其它方法向 URL 发送请求。这些方法包括 `Loader.load()` 和 `Sound.load()` 等加载方法以及 `URLLoader.load()` 和 `URLStream.load()` 等数据加载方法。

## 上载和下载文件

`FileReference.upload()` 方法可以将用户选择的文件上载到远程服务器。必须先调用 `FileReference.browse()` 或 `FileReferenceList.browse()` 方法，然后再调用 `FileReference.upload()` 方法。

调用 `FileReference.download()` 方法可打开一个对话框，用户可以在该对话框中从远程服务器下载文件。




如果服务器要求用户身份验证，则只有在浏览器中运行的 SWF 文件 （即使用浏览器插件或 ActiveX 控件的文件）才可以提供对话框来提示用户输入用户名和密码以进行身份验证，并且只适用于下载。 Flash Player 不允许上载到需要用户身份验证的服务器。

如果执行调用的 SWF 文件位于只能与本地文件系统内容交互的沙箱中，则不允许执行上载和下载操作。

默认情况下， SWF 文件不会在自身所在服务器之外的服务器上执行上载或下载操作。如果其它服务器提供跨域策略文件向执行调用的 SWF 文件所在的域授予访问权限，则执行调用的 SWF 文件可以在其它服务器上执行上载或下载操作。

# 从导入到安全域的 SWF 文件加载嵌入内容

当加载 SWF 文件时，可以设置用于加载文件的 Loader 对象的 `load()` 方法中的 `context` 参数。此参数是一个 LoaderContext 对象。将此 LoaderContext 对象的 `securityDomain` 属性设置为 `Security.currentDomain` 时， Flash Player 将在被加载 SWF 文件所在的服务器上检查是否存在跨域策略文件。如果存在跨域策略文件，并且该文件向执行加载的 SWF 文件所在的域授予访问权限，则可以作为导入媒体加载 SWF 文件。这样，执行加载的文件可以获得对 SWF 文件的库中对象的访问权限。

SWF 文件访问其它安全沙箱中被加载 SWF 文件的类的另一种方法是：使被加载的 SWF 文件调用 `Security.allowDomain()` 方法，以向执行调用的 SWF 文件所在的域授予访问权限。可以将对 `Security.allowDomain()` 方法的调用添加到被加载 SWF 文件的主类的构造函数方法中，然后使执行加载的 SWF 文件添加事件侦听器，以便响应由 Loader 对象的 `contentLoaderInfo` 属性调度的 `init` 事件。当调度此事件时，被加载的 SWF 文件已经调用构造函数方法中的 `Security.allowDomain()` 方法，因此被加载 SWF 文件中的类可用于执行加载的 SWF 文件。执行加载的 SWF 文件可以通过调用 `Loader.contentLoaderInfo.applicationDomain.getDefinition()` 从被加载的 SWF 文件中检索类。

# 处理旧内容

在 Flash Player 6 中，用于某些 Flash Player 设置的域基于 SWF 文件所在的域的末尾部分。这些设置包括对摄像头和麦克风访问权限、存储配额及永久共享对象存储的设置。

如果 SWF 文件所在的域包含的段数超过两个（如 www.example.com），则会去除该域的第一段 (www)，并使用该域的剩余部分。因此，在 Flash Player 6 中， www.example.com 和 store.example.com 都使用 example.com 作为这些设置的域。同样，www.example.co.uk 和 store.example.co.uk 都使用 example.co.uk 作为这些设置的域。这样会导致出现问题，使得来自不相关域（如 example1.co.uk 和 example2.co.uk）的 SWF 文件可以访问相同的共享对象。

在 Flash Player 7 和更高版本中，默认情况下会根据 SWF 文件所在的精确域来选择播放器设置。例如，来自 www.example.com 的 SWF 文件会对 www.example.com 使用一组播放器设置，而来自 store.example.com 的 SWF 文件会对 store.example.com 使用单独的一组播放器设置。

在使用 ActionScript 3.0 编写的 SWF 文件中，当 `Security.exactSettings` 设置为 `true`（默认值）时， Flash Player 将针对精确域使用播放器设置。当设置为 `false` 时， Flash Player 将使用 Flash Player 6 中所用的域设置。如果要更改 `exactSettings` 的默认值，则必须在需要 Flash Player 选择播放器设置的任何事件（例如使用摄像头或麦克风，或者检索永久共享对象）发生之前进行更改。

如果发布了版本 6 的 SWF 文件并通过该版本创建了永久共享对象，则要从使用 ActionScript 3.0 编写的 SWF 中检索这些永久共享对象，必须先将 `Security.exactSettings` 设置为 `false`，然后再调用 `SharedObject.getLocal()`。

# 设置 LocalConnection 权限

使用 LocalConnection 类可以开发可以互相发送指令的 SWF 文件。LocalConnection 对象只能在运行于同一台客户端计算机上的 SWF 文件之间通信，但这些 SWF 文件可以在不同的应用程序中运行。例如，一个 SWF 文件在浏览器中运行，而一个 SWF 文件在放映文件中运行。

对于每一次 LocalConnection 通信，都存在一个发送方 SWF 文件和一个侦听器 SWF 文件。默认情况下，Flash Player 允许在同一域中的 SWF 文件之间进行 LocalConnection 通信。对于不同沙箱中的 SWF 文件，侦听器必须通过使用 `LocalConnection.allowDomain()` 方法来允许发送方具有访问权限。作为参数传递到 `LocalConnection.allowDomain()` 方法的字符串可以包含以下任意项：确切域名、 IP 地址和通配符 *。

> **注意** `allowDomain()` 方法的格式已更改，与其在 ActionScript 1.0 和 2.0 中的格式不同。在这两个早期版本中， `allowDomain()` 是可以实现的回调方法。在 ActionScript 3.0 中，`allowDomain()` 则是调用的 LocalConnection 类的内置方法。由于此更改，`allowDomain()` 的用法与 `Security.allowDomain()` 基本相同。

SWF 文件可以使用 LocalConnection 类的 `domain` 属性确定其所在的域。

# 控制对主机网页中脚本的访问

通过使用以下 ActionScript 3.0 API 可实现外出脚本访问：

- `flash.system.fscommand()` 函数
- `flash.net.navigateToURL()` 函数 （当指定 `navigateToURL("javascript: alert('Hello from Flash Player.')"` 等脚本访问语句时）
- `flash.net.navigateToURL()` 函数 （当 `window` 参数设置为 “`_top`”、“`_self`” 或 “`_parent`” 时）
- `ExternalInterface.call()` 方法

对于本地运行的 SWF 文件，仅当 SWF 文件和包含该文件的网页 （如果存在）位于受信任的本地安全沙箱中时，才能成功调用这些方法。如果内容位于只能与远程内容交互的沙箱或只能与本地文件系统内容交互的沙箱中，则对这些方法的调用将失败。

HTML 代码中用于加载文件的 `AllowScriptAccess` 参数控制能否从 SWF 文件内执行外出脚本访问。

在 HTML 代码中为承载 SWF 文件的网页设置此参数。可以在 `PARAM` 或 `EMBED` 标签中进行设置。

`AllowScriptAccess` 参数可以有 `"always"`、 `"sameDomain"` 和 `"never"` 这三个可能值中的一个：

- 当 `AllowScriptAccess` 为 “`sameDomain`” 时，仅当 SWF 文件和网页位于同一域中时才允许执行外出脚本访问。这是 AVM2 内容的默认值。
- 当 `AllowScriptAccess` 为 “`never`” 时，外出脚本访问将始终失败。
- 当 `AllowScriptAccess` 为 “`always`” 时，外出脚本访问将始终成功。

如果未在 HTML 页面中为 SWF 文件指定 `AllowScriptAccess` 参数，则默认为 AVM2 内容的 “`sameDomain`”。

下面是一个在 HTML 页面中设置 `AllowScriptAccess` 标签的示例：

```
<object id='MyMovie.swf' classid='clsid:D27CDB6E-AE6D-11cf-96B8-
  444553540000' codebase='http://download.adobe.com/pub/shockwave/cabs/
  flash/swflash.cab#version=9,0,0,0' height='100%' width='100%'>
<param name='AllowScriptAccess' value='never'/>
<param name='src' value=''MyMovie.swf'/>
<embed name='MyMovie.swf' pluginspage='http://www.adobe.com/go/
  getflashplayer_cn' src='MyMovie.swf' height='100%' width='100%'
  AllowScriptAccess='never'/>
</object>
```

`AllowScriptAccess` 参数可以防止从一个域中承载的 SWF 文件访问来自另一个域的 HTML 页面中的脚本。对从另一个域承载的所有 SWF 文件使用 `AllowScriptAccess="never"` 可以确保位于 HTML 页面中的脚本的安全性。

有关详细信息，请参阅 《ActionScript 3.0 语言和组件参考》中的以下条目：

- `flash.system.fscommand()` 函数
- `flash.net.navigateToURL()` 函数
- ExternalInterface 类的 `call()` 方法

# 共享对象

Flash Player 提供使用 "共享对象" 的功能，这些对象是永久位于 SWF 文件外部的 ActionScript 对象，它们或者位于用户的本地文件系统中，或者位于远程 RTMP 服务器上。共享对象与 Flash Player 中的其它媒体相似，也划分到安全沙箱中。但是，共享对象的沙箱模型稍有不同，因为共享对象不是可以跨域边界访问的资源，而是始终从共享对象存储区获得的资源，该存储库特定于调用 SharedObject 类的方法的每个 SWF 文件的域。通常，共享对象存储区比 SWF 文件所在的域更精确：默认情况下，每个 SWF 文件使用特定于其整个源 URL 的共享对象存储区。

SWF 文件可以使用 `SharedObject.getLocal()` 和 `SharedObject.getRemote()` 方法的 `localPath` 参数，以便使用仅与其部分 URL 关联的共享对象存储区。这样， SWF 文件可以允许与其它 URL 的其它 SWF 文件共享。即使将 `'/'` 作为 `localPath` 参数传递，仍然会指定特定于其自身所在域的共享对象存储区。

用户可通过使用 "Flash Player 设置" 对话框或 "设置管理器" 来限制共享对象访问。默认情况下，可以将共享对象创建为每个域最多可以保存 100 KB 的数据。管理用户和用户还可限制写入文件系统的能力。有关详细信息，请参阅第 652 页的 "管理用户控制"和第 654 页的 "用户控制"。

可以通过为 SharedObject.getLocal() 方法或 SharedObject.getRemote() 方法的 secure 参数指定 true 来指定共享对象是安全的。请注意有关 secure 参数的以下说明：

- 如果此参数设置为 true，则 Flash Player 将创建一个新的安全共享对象或获取一个对现有安全共享对象的引用。此安全共享对象只能由通过 HTTPS 传递的 SWF 文件来读取或写入，SWF 文件将调用 SharedObject.getLocal() 并将 secure 参数设置为 true。
- 如果此参数设置为 false，则 Flash Player 将创建一个新的共享对象或获取一个对现有共享对象的引用，后者可由通过非 HTTPS 连接传递的 SWF 文件来读取或写入。

如果执行调用的 SWF 文件不是来自 HTTPS URL，则为 SharedObject.getLocal() 方法或 SharedObject.getRemote() 方法的 secure 参数指定 true 会导致 SecurityError 异常。

对共享对象存储区的选择基于 SWF 文件的源 URL。即使在导入加载和动态加载这两种情况下也是如此，这时 SWF 文件不是源自简单的 URL。导入加载是指在 LoaderContext.securityDomain 属性设置为 SecurityDomain.currentDomain 时加载 SWF 文件。在这种情况下，被加载的 SWF 文件将具有一个伪 URL，该 URL 以执行加载的 SWF 文件所在的域开头，后跟实际的源 URL。动态加载是指使用 Loader.loadBytes() 方法加载 SWF 文件。在这种情况下，被加载的 SWF 文件将具有一个伪 URL，该 URL 以执行加载的 SWF 文件的完整 URL 开头，后跟一个整数 ID。在导入加载和动态加载这两种情况下，都可以使用 LoaderInfo.url 属性检查 SWF 文件的伪 URL。选择共享对象存储区时，可将该伪 URL 完全视为真实的 URL。可以指定使用部分或全部伪 URL 的共享对象 localPath 参数。

用户和管理员可以选择禁止使用“第三方共享对象”。当在 Web 浏览器中执行的任何 SWF 文件的源 URL 与浏览器地址栏中显示的 URL 属于不同的域时，该 SWF 文件可以使用这样的共享对象。用户和管理员可以出于隐私原因选择禁止使用第三方共享对象，从而可以避免跨域跟踪。为避开这种限制，可能希望确保使用共享对象的任何 SWF 文件都只在 HTML 页面结构内部进行加载，这样可以确保 SWF 文件位于浏览器的地址栏中所示的同一个域中。试图使用第三方 SWF 文件中的共享对象时，如果第三方共享对象禁止使用，则 SharedObject.getLocal() 和 SharedObject.getRemote() 方法将返回 null。有关详细信息，请访问 www.adobe.com/products/flashplayer/articles/thirdpartylso。

# 摄像头、麦克风、剪贴板、鼠标和键盘访问

当 SWF 文件试图使用 Camera.get() 或 Microphone.get() 方法访问用户的摄像头或麦克风时，Flash Player 将显示一个“隐私”对话框，用户可以在该对话框中允许或拒绝对其摄像头或麦克风的访问。用户和管理用户还可以通过 mms.cfg 文件、“设置 UI”和“设置管理器”中的控制基于站点或全局禁用摄像头访问（请参阅第 652 页的“管理用户控制”和第 654 页的“用户控制”）。用户加以限制后，Camera.get() 和 Microphone.get() 方法均返回 null 值。可以使用 Capabilities.avHardwareDisable 属性确定是已经通过管理方式禁止了 (true) 对摄像头和麦克风的访问，还是允许 (false) 对摄像头和麦克风的访问。

`System.setClipboard()` 方法允许 SWF 文件用纯文本字符串替换剪贴板内容。这不会带来任何安全性风险。为避免因剪切或复制到剪贴板的密码和其它敏感数据所带来的风险，并未提供相应的 “getClipboard”（读取）方法。

Flash 应用程序仅可以监视在其焦点以内发生的键盘和鼠标事件，无法检测其它应用程序中的键盘或鼠标事件。

# 索引

## 符号



## 英文

## D



## E



## F

## K



## L

## M



## N



## P



## Q

T

## W

## X

# Y



# Z